Ein transformativer Ansatz für die Synthese und Verifikation algorithmischer Hardwarebeschreibungen

Vom Fachbereich
Elektrotechnik und Informationstechnik der
Technischen Universität Darmstadt
zur Erlangung der Würde eines
Doktor Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

Holger Hinrichsen
geboren 29.09.1970

Referent:	Prof. Dr.-Ing. H. Eveking
Korreferent:	Prof. Dr.-Ing. R. Hoffmann

Tag der Einreichung:	21.04.2000
Tag der mündlichen Prüfung:	04.12.2000

D17
Darmstädter Dissertation

Zusammenfassung

In dieser Arbeit wird ein Verfahren der formal korrekten Synthese vorgestellt, mit Hilfe dessen ein automatisierter Entwurf von Prozessoren mit Pipelining möglich ist. Das Verfahren basiert auf einer kleinen Menge korrektheitserhaltender Transformationen, deren Anwendung effizient durch eine unabhängige Post-Synthese-Verifikation überprüft wird. Zur Spezifikation dient eine im Rahmen dieser Arbeit entwickelte, experimentelle Hardwarebeschreibungssprache *LLS (Language of Labelled Segments)*.
Der transformale Ansatz kann neben der Synthese auch zur Verifikation genutzt werden. Ein automatisches Verfahren wird vorgestellt, daß die Äquivalenz eines Entwurfs vor und nach dem Einplanungsschritt der High-Level-Synthese nachweist. Durch Anwenden von Transformationen kann die Äquivalenz zweier Beschreibungen gezeigt werden. Die Äquivalenz ist gegeben, wenn eine Folge korrektheitserhaltender Transformationen existiert, die die eine in die andere Beschreibung umwandelt. Insbesondere Ergebnisse moderner Einplanungsverfahren wie z.B. AFAP und DLS können mit dem vorgestellten Verfahren, das sowohl auf zyklischen Kontrollflußgraphen als auch für Pipeline-Systeme arbeitet, verifiziert werden.

Schlüsselworte

Hardwaresynthese, formale Synthese, formale Verifikation

Abstract

A method of formally correct synthesis is presented and applied to the automatic construction of pipelined processors. The approach is based on a small set of correctness-preserving transformations that are efficiently cross-checked by an independent formal verification tool. For specification an experimental hardware description language *LLS (Language of Labelled Segments)* is used.
The transformational technique can also be used for verification. A method for the fully automatic equivalence verification of a design before and after the scheduling step of high-level synthesis is presented. The equivalence of two descriptions is given if a sequence of transformations exists which turns the first description into the second. The technique is applicable to the results of advanced scheduling methods like AFAP and DLS, which work on cyclic control flows, as well as to pipelined designs.

Keywords

hardware synthesis, formal synthesis, formal verification

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Heutzutage erlauben moderne Synthesewerkzeuge die Synthese hochgradig komplexer Schaltungen. Der Entwurfsablauf wird mehr und mehr automatisiert, so daß für den Entwerfer der Ablauf der Synthese und die erzielte Implementierung immer weniger zu überblicken sind. Es wird somit für den Entwickler immer schwieriger, die Korrektheit der abgeleiteten Implementierung bezüglich der vorgegebenen Spezifikation zu überprüfen (z.B. Intel's Pentium Bug). Der Nachweis der Korrektheit ist jedoch erforderlich, da Fehler der Synthesewerkzeuge aufgrund deren Komplexität nicht auszuschließen sind.
Grundsätzlich können hierbei drei Ansätze unterschieden werden, die die Korrektheit einer Synthese sicherstellen. Zum einen ist es möglich, das Synthesewerkzeug formal zu verifizieren. Die Komplexität der Programme ist jedoch in den letzten Jahren stark gestiegen, da immer größere Schaltungen synthetisiert und zunehmend abstraktere Beschreibungssprachen unterstützt werden. Die Verifikation solcher aufwendigen Synthesewerkzeuge, die nicht selten aus mehreren 100.000 Programmzeilen bestehen, ist im allgemeinen nicht möglich.
Zum anderen kann das Ergebnis formal verifiziert werden. Dieser Ansatz wird in vielen Bereichen bereits eingesetzt, bringt aber auch einige Schwierigkeiten mit sich. Probleme bereiten einerseits die Komplexität der zu verifizierenden Beschreibungen, andererseits auftretende Zyklen. Die Äquivalenz von Spezifikation und Implementierung wird im allgemeinen gezeigt, indem endliche Sequenzen betrachtet werden. Aus diesem Grund können z.B. Ergebnisse von modernen Einplanungsverfahren oftmals nicht verifiziert werden.
Ein dritter Ansatz besteht darin, die Synthese auf die Anwendung von korrektheitserhaltenden Transformationen einzuschränken. Als Ergebnis ergibt sich nicht nur eine Implementierung, sondern auch ein Beweis deren Korrektheit in Bezug auf die vorgegebene Spezifikation. Obwohl in vielen Synthesewerkzeugen intern Transformationen verwendet werden, existieren nur sehr wenige interaktive, transformationsbasierte Systeme. In dieser Arbeit wird ein Transformationswerkzeug vorgestellt, das auf einer Menge allgemeingültiger, formal korrekter

Transformationen beruht. Ein Benutzer kann aus einer Menge von Axiomen und Regeln eine Transformation auswählen und auf eine Beschreibung anwenden. Während andere Verfahren z.B. durch Retiming die Register-Transfer-Struktur eines Systems transformieren, arbeitet der hier dargestellte Ansatz auf algorithmischen Beschreibungen.
Soll ein Transformationsprozeß automatisiert werden, können die Transformationen zu sogenannten *Meta-Transformationen* kombiniert werden. Als Beispiel wird ein Verfahren zur automatisierten Einplanung von Pipelines vorgestellt. Ausgehend von der Beschreibung eines sequentiellen Prozessors wird durch sukzessives Anwenden von Transformationen eine Implementierung als Pipeline abgeleitet. Das Verfahren wurde unter anderem angewendet, um eine DLX-Pipeline [HP96] einzuplanen. Es zeigte sich, daß das automatisch eingeplante Ergebnis den manuell erzeugten Entwürfen entsprach.
Eine weitere Anwendung des Transformationswerkzeugs stellt die Verifikation von Einplanungsverfahren dar. Das Ergebnis der Einplanungsphase kann unter Anwendung von Transformationen auf Äquivalenz mit der ursprünglichen Spezifikation getestet werden.

Motivation der Arbeit

Pipelining ist eine bekannte Technik um die Ausführung eines Programmes zu beschleunigen, indem die Befehle nicht nacheinander, sondern teilweise parallel abgearbeitet werden. Die Zeit, die für die Ausführung eines Befehls benötigt wird, bleibt gleich, der Durchsatz erhöht sich. Die automatisierte Einplanung eines Pipelinesystems ist schwierig, da bei der Ablaufplanung mögliche Datenabhängigkeiten und Konflikte berücksichtigt werden müssen. Diese Schwierigkeiten ergeben sich aus der überlappenden Ausführung mehrerer Befehle in der Pipeline. Mögliche Datenabhängigkeiten und Konflikte können zwar bereits der Spezifikation entnommen werden, doch stellt es sich als sehr schwierig heraus, wirklich alle möglichen Konflikte zu entdecken. Insbesondere für das sehr komplexe Pipelining von Prozessoren sind daher bisher nur sehr wenige Verfahren entwickelt worden.

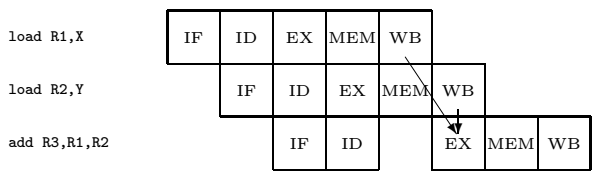


Abb. 1.1: Datenabhängigkeiten bei der Ausführung von Befehlen

Führt z.B. der DLX-Prozessor [HP96] einen Additions-Befehl aus, dem zwei

Lade-Befehle vorausgehen, müssen verschiedene Fälle bedacht werden. Abbildung 1.1 illustriert das Beispiel. Während die Pfeile Datenabhängigkeiten andeuten, symbolisieren `IF` bis `WB` verschiedene Phasen bei der Abarbeitung eines Befehls. Übereinander gezeichnete Phasen werden parallel, hintereinander angeordnete nacheinander ausgeführt. Abhängigkeiten zwischen der ersten Lade- und der Additions-Instruktion erfordern sogenannte *Forwarding-Techniken*, so daß ein Anhalten der Pipeline vermieden werden kann. Besteht zwischen dem zweiten Lade- und dem Additions-Befehl eine Adressengleichheit, muß die Bearbeitung des Additions-Befehls für einen Schritt ausgesetzt werden. In diesem Fall muß eine Abhängigkeit zwischen dem ersten Lade- und dem Additions-Befehl auf eine andere Art gelöst werden. Somit muß beim Lösen des Konflikts zwischen der ersten Lade- und der Additions-Instruktion berücksichtigt werden, ob ein weiterer Konflikt zwischen dem zweiten Lade- und dem Additions-Befehl besteht (in Kapitel 5 wird das Beispiel näher erläutert).

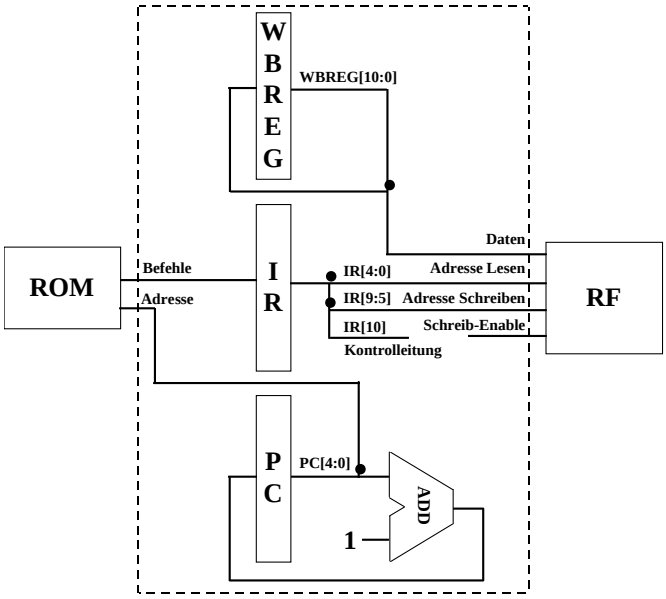


Abb. 1.2: Datenpfad eines sequentiellen Prozessors

Abbildung 1.2 stellt als weiteres Beispiel den Datenpfad eines sehr einfachen sequentiellen Prozessors dar. Das Befehlsregister `IR` lädt aus dem Speicher `ROM` die jeweilig auszuführende Instruktion (**`Instruction-Fetch`**-Phase). Die Befehle werden hierbei über den Befehlszähler `PC` adressiert. Ist das höchstwertige Bit eines Befehls eine `'1'`, wird im nächsten Schritt ein Datenwort aus einem Register der Registerbank `RF` nach `WBREG` ausgelesen (`Read`-Phase) und im anschließenden Takt wieder in `RF` abgespeichert (`Write`-Phase). Die Adressierung erfolgt durch `IR`. Ein Befehl besteht somit aus drei Phasen. Die Ausführung benötigt drei Zeitschritte. Ist das höchstwertige Bit eines Befehls eine `'0'`, so wird ein Leerschritt (`STALL`) vollzogen, so daß die Ausführung zwei Zeitschritte benötigt.
Abbildung 1.3 stellt das Ablaufdiagramm des Pipelinesystems dar, während der Datenpfad des Prozessors mit Pipelining Abbildung 1.4 zu entnehmen ist. Der

sequentielle Prozessor benötigt zur Ausführung der Sequenz in Abbildung 1.3 elf Zeitschritte, das Pipelinesystem nur sechs.

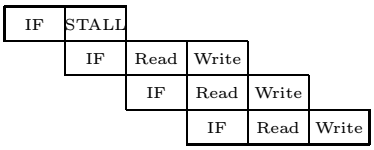


Abb. 1.3: Ablaufdiagramm der dreistufigen Pipeline

Durch Anwenden von 18 logischen Transformationen kann aus der sequentiellen Spezifikation eine Pipeline-Implementierung abgeleitet werden. Das Ergebnis zeigt, daß zwei Datenabhängigkeiten zu beachten sind. Zum einen wird ein weiteres Register `P`, ein sogenanntes *Pipeline-Register* benötigt, das den Befehl aus `IR` speichert, da in jedem Schritt eine neue Instruktion geladen wird. Zum anderen muß durch *Forwarding* der Wert eines Lesevorgangs bei Adressengleichheit übernommen werden. Speichert eine Instruktion den Wert aus `WBREG` genau in das Register der Registerbank `RF`, aus dem ein nachfolgender Befehl den Wert auslesen möchte, wird der Lesezugriff nicht durchgeführt, da der Wert noch nicht in `RF` verfügbar ist, aber bereits in `WBREG` gespeichert ist.

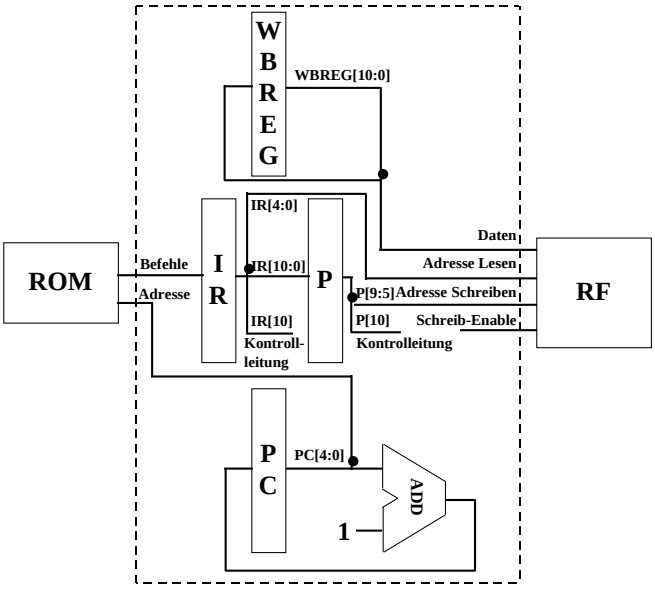


Abb. 1.4: Datenpfad eines Prozessors mit Pipelining

In dieser Arbeit wird ein Transformationswerkzeug vorgestellt, das ausgehend von einer sequentiellen Spezifikation durch sukzessives Anwenden von Transformationen ein Pipelinesystem ableitet. Das Werkzeug ermöglicht es, interaktiv den Entwurfsraum zu evaluieren. Bereits kleine Veränderungen des Befehlssatzes, wie z.B. das Abwandeln oder Verschieben einer Operation, haben einen großen Einfluß auf die Art und die Anzahl der Konfliktfälle in einer Pipeline. Das Werkzeug

findet alle möglichen Konflikte, die durch *Forwarding*, Einführen von *Pipeline-Registern* oder das Anhalten der Pipeline gelöst werden, und plant eine dynamische Pipeline ein (das Verfahren wird in Kapitel 5 dargestellt).
Aufgrund der oben genannten Schwierigkeiten, die sich bei der Einplanung einer Pipeline ergeben, ist die Frage nach der Korrektheit einer Lösung essentiell. Da die Beschreibungen zyklisch sind, ist die Verifikation der eingeplanten Pipeline schwierig. Der Nachweis kann nur dadurch erfolgen, daß finite Folgen von Berechnungen der Spezifikation und des Pipelinesystems verglichen werden. Ein transformativer Ansatz umgeht diese Schwierigkeiten, da die Korrektheit des Ergebnisses über die Folge der während der Synthese angewendeten Transformationen gegeben ist. Die Korrektheit des Ergebnisses ergibt sich somit durch seine Konstruktion. Die Äquivalenz der Spezifikation und des Pipelinesystems wird in diesem Ansatz nur noch überprüft, um Implementierungsfehler des Transformationswerkzeugs zu finden (Details werden in Kapitel 5 ausführlich beschrieben).
Die Pipeline-Synthese ist aber nur ein Anwendungsbeispiel für das in dieser Arbeit vorgestellte Transformationswerkzeug. Durch Anwendung von Transformationen kann ein Benutzer für eine Spezifikation eine Implementierung ableiten. Die Transformationen erhalten die Korrektheit hinsichtlich der ursprünglichen Beschreibung. Wie in der Einleitung bereits erwähnt, kann das Transformationswerkzeug darüberhinaus auch zur Verifikation eingesetzt werden.

Gliederung der Arbeit

In dieser Arbeit wird das *TUD Transformationswerkzeug (TUDT)* vorgestellt, das es ermöglicht, algorithmische Beschreibungen formal korrekt zu verändern. Einen Überblick über den formal korrekten Hardwareentwurf gibt Kapitel 2. Bereits existierende, transformative Synthesewerkzeuge werden vorgestellt und mit dem in dieser Arbeit dargestellten Ansatz verglichen. Abschließend wird das Transformationswerkzeug in den Entwurfsfluß eingeordnet.
Als Eingabesprache des Werkzeugs dient die im Rahmen dieser Arbeit entwickelte, experimentelle Hardwarebeschreibungssprache *LLS (Language of Labelled Segments)*. In Kapitel 3 wird LLS vorgestellt, eine Normalform einer LLS-Beschreibung definiert und deren Semantik angegeben. Zum Abschluß des Kapitels wird dargestellt, wie eine LLS-Beschreibung nach VHDL übersetzt werden kann.
Kapitel 4 stellt das Transformationssystem und die mathematischen Grundlagen dar. Zwei Klassen von Transformationen werden vorgestellt, die es ermöglichen den Kontrollfluß und/oder den Datenfluß einer Beschreibung formal korrekt zu verändern. Zu Beginn des Kapitels werden die entsprechenden Äquivalenz-Begriffe definiert. Betrachtungen über die Vollständigkeit des Transformationssystems beschließen das Kapitel. Bei Kenntnis der Kapitel 3 und 4 können die

übrigen Kapitel auch unabhängig voneinander gelesen werden.
Als Anwendungsbeispiel wird in Kapitel 5 ein Verfahren zur Pipeline-Synthese vorgestellt. Ausgehend von einem sequentiellen Prozessor wird durch sukzessives Anwenden von Transformationen eine Pipeline-Implementierung abgeleitet. Als Beispiele dienen unter anderem ein DLX- und ein PIC-Prozessor. Um Implementierungsfehler auszuschließen, wird die Korrektheit der Ergebnisse durch ein unabhängiges Werkzeug der formalen Verifikation nachgewiesen.
In Kapitel 6 wird ein Ansatz zur Verifikation von Einplanungsverfahren vorgestellt, der zeigt, daß das Transformationswerkzeug ebenfalls zur Verifikation eingesetzt werden kann. Die Äquivalenz zweier Beschreibungen kann gezeigt werden, indem durch Anwenden von Transformationen die eine Beschreibung in die andere umgeformt wird.
In Kapitel 7 werden Methoden dargestellt, mit denen algorithmische Beschreibungen effizient vereinfacht werden können. Diese Vereinfachungstechniken werden benötigt, um eine automatisierte Einplanung für Prozessoren mit Pipelining zu ermöglichen, da in einigen Fällen die Durchführung ohne Anwendung von Minimierungstechniken unmöglich wird. Die Techniken dienen dazu, logisch unmögliche Pfade zu eliminieren und die Kontrollstrukturen zu vereinfachen.
Einen Überblick über die Implementierung des Transformationswerkzeugs gibt Kapitel 8. Abschließend faßt Kapitel 9 die erzielten Ergebnisse zusammen und gibt einen Ausblick.

Kapitel 2

Transformative Ansätze der Synthese und der Verifikation

2.1 Einleitung

Ein Syntheseﬂuß ist formal korrekt, falls die Synthese von einer Verifikation begleitet wird. Es besteht die Möglichkeit, entweder die Korrektheit des Werkzeugs nachzuweisen (Prä-Synthese-Verifikation), die einzelnen Schritte der Synthese zu überprüfen (formale Synthese) oder die Äquivalenz der ursprünglichen Spezifikation und des Syntheseergebnisses zu zeigen (Post-Synthese-Verifikation).
In dieser Arbeit wird das *TUD Transformationswerkzeug (TUDT)* vorgestellt. Als Werkzeug der formalen Synthese unterstützt es den formal korrekten Hardwareentwurf. Wird die Synthese auf das Anwenden von korrektheitserhaltenden Transformationen beschränkt, ist die Korrektheit des Syntheseergebnisses durch die Folge der angewendeten Transformationen gegeben. Der formale Ansatz erlaubt es, bei komplizierten Problemen wie z.B. der Einplanung von Befehlsabläufen in Prozessoren ein Lösungsverfahren zu entwickeln. Implementierungsfehler des Transformationswerkzeugs können zwar nicht ausgeschlossen werden, sie können aber durch Einbeziehen von Methoden der formalen Verifikation gefunden werden.
Darüberhinaus kann das Werkzeug auch zur Äquivalenzprüfung eingesetzt werden. Zwei Beschreibungen sind äquivalent, wenn sie durch Anwendung von Transformationen aneinander angeglichen werden können. Im Gegensatz zu den bekannten Ansätzen, die nur endliche Sequenzen von Anweisungen auf Äquivalenz überprüfen, kann das vorgestellte Verfahren auch zur Äquivalenzprüfung von zyklischen Beschreibungen eingesetzt werden.
Während Abschnitt 2.2 die verschiedenen Möglichkeiten des formal korrekten Hardwareentwurfs erläutert, stellt Abschnitt 2.3 verschiedene Synthesewerkzeuge vor, die einen transformativen Ansatz verfolgen. Der Einsatz transformativer

Techniken zur Verifikation wird in Abschnitt 2.4 dargestellt. Abschnitt 2.5 stellt das *TUD Transformationswerkzeug (TUDT)* vor und zieht einen Vergleich mit bestehenden Ansätzen. Abschnitt 2.6 ordnet das Transformationswerkzeug in den Entwurfsfluß ein. Eine Zusammenfassung beendet das Kapitel.

2.2 Synthese korrekter Schaltungen

Synthese bezeichnet den Vorgang, aus einer gegebenen Spezifikation eine Implementierung abzuleiten. Es ist ein komplexer Vorgang, der sich aus mehreren Teilschritten auf verschiedenen Entwurfsebenen zusammensetzt. Auf der *Systemebene* wird das zu modellierende System durch miteinander kommunizierende Komponenten wie z.B. Prozessoren, Speicher u.a. beschrieben. Das Verhalten der Komponenten wird auf der darunter liegenden *algorithmischen* bzw. *Verhaltensebene* definiert. Der Aufbau einer Komponente durch Register, logisch algorithmische Einheiten u.a. sowie deren Verbindungen untereinander werden auf der *Register-Transfer-Ebene* angegeben. Auf der *Gatterebene* werden die Komponenten der Register-Transfer-Ebene durch die grundlegenden logischen Gatter wie AND, OR, NOT, NAND, NOR und Flip-Flops dargestellt. Die elektrischen und fertigungstechnischen Eigenschaften werden schließlich auf der *Schalter-, Layout- und Prozeßebene* festgelegt. Die angegebene Taxonomie richtet sich nach [Mar93]. Das in dieser Arbeit vorgestellte Transformationswerkzeug arbeitet sowohl auf der algorithmischen als auch auf der Register-Transfer-Ebene.
Der formal korrekte Hardwareentwurf [Eve90] fordert eine die Synthese begleitende Verifikation der Ergebnisse. Verifikation meint den Nachweis, daß die Anforderungen, die die Spezifikation erhebt, durch die Implementierung erfüllt werden. Es können drei Ansätze unterschieden werden [KBES96], die sich durch den Zeitpunkt, zu dem der Korrektheitsbeweis gegeben wird, unterscheiden:

- Die *Prä-Synthese-Verifikation* weist die Korrektheit des Syntheseverfahrens nach, so daß der Korrektheitsbeweis *vor* der Synthese erfolgt.
- Die *formale Synthese* schränkt die Synthese auf die Anwendung von korrektheitserhaltenden Transformationen ein und gibt somit einen Beweis für die Korrektheit *während* der Synthese.
- Die *Post-Synthese-Verifikation* verifiziert das Syntheseergebnis erst *nach* Durchführung der Synthese.

2.2.1 Prä-Synthese-Verifikation

Die Prä-Synthese-Verifikation zielt auf eine formale Verifikation der Synthesealgorithmen. Unter Anwendung von Methoden der Programmverifikation wird

gezeigt, daß ein Syntheseverfahren für alle möglichen Eingaben korrekte Ergebnisse erzeugt. Die Korrektheit ist im Unterschied zu der formalen Synthese und der Post-Synthese-Verifikation, bei denen der Korrektheitsbeweis für jedes Syntheseergebnis erbracht werden muß, nur einmal zu zeigen.
Beispiele sind der Korrektheitsbeweis des *Force Directed List Scheduling (FDLS)* [PK89b] Algorithmus in [NTR+98] oder die Verifikation eines Register-Allokationsverfahrens in [NV98]. Die Verifikation erfolgt unter Verwendung eines Theorembeweisers. Beide Ansätze arbeiten mit einem vergleichsweise abstrakten Modell des Synthesealgorithmus, so daß ein Schutz gegen Fehler der programmtechnischen Implementierung des Synthesewerkzeugs nicht gegeben ist.
Die Korrektheit des Synthesewerkzeugs *PBS (Proven Boolean Simplification)* [AL91] konnte ebenfalls mit Hilfe eines Theorembeweisers nachgewiesen werden. PBS führt eine rein boolesche Optimierung durch. Es stellt ein einfaches Syntheseverfahren dar, das nicht mit kommerziell verfügbaren vergleichbar ist.

2.2.2 Formale Synthese

Unter formaler Synthese werden Methoden verstanden, die das Syntheseergebnis innerhalb eines logischen Kalküls formal korrekt ableiten. Die Synthese ist begrenzt auf die Anwendung formal definierter, korrektheitserhaltender Transformationen, so daß nicht nur eine Implementierung für eine vorgegebene Spezifikation gewonnen wird, sondern gleichfalls ein Beweis für die Korrektheit des Ergebnisses gegeben werden kann.
Der Kern der Synthesewerkzeuge besteht aus einer Menge von Transformationsregeln. Dabei kann zwischen hardware-spezifischen und allgemeingültigen bzw. logischen Transformationen unterschieden werden [KBES96]. Bei hardware-spezifischen Kalkülen können beliebige Datenstrukturen zur Schaltungsrepräsentation eingesetzt werden. Für die Implementierung der elementaren Schaltungstransformationen gibt es keine Beschränkung, so daß der Einsatz von hardwarespezifischen Kalkülen im allgemeinen zu effizienten Verfahren führt. Gleichzeitig enthalten hardware-spezifische Kalküle eine große Zahl komplexer Regeln, deren programmtechnische Implementierung sicherheitskritisch für die Richtigkeit der Korrektheitsaussage eines Systems ist.
Die Verwendung von logischen Kalkülen bedeutet, daß alle elementaren Transformationen auf logische Regelanwendungen zurückgeführt werden müssen. Der Kern eines Werkzeugs umfaßt im allgemeinen eine Bibliothek mit nur wenigen Regeln, die unabhängig von einer bestimmten Anwendung Gültigkeit besitzen. Einerseits ist damit eine breitere Anwendbarkeit des Werkzeugs gegeben, andererseits erhöht sich auch die Sicherheit gegenüber Fehlern bei der Implementierung. Der Transformationsprozeß hingegen verlängert sich in der Regel.
Viele formale Synthesewerkzeuge wie z.B. *LAMBDA/DIALOG* [FFFH90], *VERITAS* [HDL89] oder *HASH* [BS99] betten den Transformationsprozeß in einen

Theorembeweiser ein. Dabei ergibt sich das Problem, daß das Synthesewissen innerhalb des Theorembeweisers rekonstruiert und formalisiert werden muß. Existierende Synthesewerkzeuge können normalerweise nicht eingesetzt werden.
Im Rahmen dieser Arbeit wird ein Verfahren der formalen Synthese vorgestellt. Der Kern des Werkzeugs besteht aus einer kleinen Menge allgemeingültiger Axiome und Regeln, was die Sicherheit gegenüber Fehlern der Implementierung erhöht. Die Korrektheit des Syntheseergebnisses wird nicht durch Verwendung eines Theorembeweisers, sondern durch Anwendung von Methoden der Post-Synthese-Verifikation sichergestellt.

2.2.3 Post-Synthese-Verifikation

Zur Zeit kommen hauptsächlich Post-Synthese-Verifikationsverfahren zum Einsatz. Die Synthese kann von herkömmlichen Werkzeugen durchgeführt werden. Die Verifikation ist unabhängig von dem Synthesevorgang und weist nach, daß die Implementierung die Anforderungen der Spezifikation erfüllt. Die verwendeten Verifikationstechniken sind im allgemeinen zeitaufwendig. Bereits bei booleschen Schaltnetzen ist diese Problemstellung NP-vollständig.
FORVERTIS (Formale Verifikation von Transformationsregeln in der Synthese digitaler Schaltungen) [Mut97, LMM98] z.B. führt eine Post-Synthese-Verifikation durch. Im Unterschied zu anderen Ansätzen werden die Anforderungen, die die Korrektheit der Synthese garantieren, durch einen Prä-Synthese-Verifikationsschritt aus der Synthesemethodik abgeleitet. In der Post-Synthese-Verifikation wird mit Hilfe des Theorembeweisers HOL [GM93] gezeigt, daß das Ergebnis die durch die Prä-Synthese-Verifikation gewonnenen Anforderungen erfüllt.
In der vorliegenden Arbeit werden die Syntheseergebnisse durch eine symbolische Simulation [REH99, RHE99a, RHE99b] überprüft, da nicht ausgeschlossen werden kann, daß die Implementierung des Transformationswerkzeugs fehlerhaft ist (siehe Abschnitt 4.9). Die symbolische Simulation ist ein Verfahren der Post-Synthese-Verifikation, bei dem Register nur symbolische Werte annehmen. Ein symbolisch simulierter Pfad einer Beschreibung entspricht somit einer Vielzahl "klassischer" Simulationsläufe, so daß eine Beschreibung vollständig simuliert werden kann. Wenn auf allen Pfaden gleiche Endergebnisse erzielt werden, sind die zu vergleichenden Beschreibungen äquivalent.

2.3 Werkzeuge der formalen Synthese

2.3.1 RUBY und T-RUBY

RUBY [JS93] wurde als eine Sprache zur Repräsentation von Schaltungsstrukturen entwickelt. Schaltungen werden in RUBY nicht wie in klassischen Ansätzen

durch Netzlisten beschrieben, sondern durch Operationen auf Komponenten definiert. Das Verhalten einer Schaltung wird durch binäre Relationen definiert. Durch Ausführen von Äquivalenzumformungen auf RUBY-Ausdrücken wird eine Implementierung abgeleitet. Hierzu wurde eine Vielzahl von Gleichungen definiert, deren Anwendung das Verhalten der zugehörigen Schaltung nicht verändert. So stehen z.B. die parallele Komposition oder Retiming als Transformationen zur Verfügung.

Abb. 2.1: Graphische Interpretation von RUBY-Strukturen

Jede Komponente besitzt auch eine graphische Interpretation, so daß eine graphische Repräsentation von RUBY-Ausdrücken möglich ist. In Abbildung 2.1 wird z.B. die sequentielle `R;S` bzw. parallele Komposition `[R,S]` zweier Komponenten `R` und `S` dargestellt.
Das System T-RUBY [SR95] stellt die Implementierung eines formalen Kalküls dar, in dem RUBY-Ausdrücke repräsentiert und transformiert werden können.

2.3.2 RLEXT

RLEXT (Register Level Exploration Tool) [Kna89, KW92] ist ein interaktives Werkzeug für die Synthese des Datenpfades auf der Register-Transfer-Ebene. Schaltungen werden im Sinne einer objektorientierten Programmierung als Objekte dargestellt, die durch Datenfluß, Zeitverhalten, Struktur und eine Menge von Beziehungen, mit denen die Relationen zwischen Datenfluß, Zeitverhalten und Struktur zu beschreiben sind, charakterisiert werden. Das Datenflußmodell repräsentiert das Verhalten eines Entwurfs, indem es die Daten, Parameter und Operationen in Beziehung zueinander setzt, während das Zeitmodell die abstrakten Zustände sowie die Zustandsübergänge eines Entwurfs darstellt. Das strukturelle Modell gibt an, welche funktionalen Einheiten, wie z.B. Register, Addierer, Multiplexer u.a., verwendet werden.
RLEXT zählt zu den Systemen, die auf einem hardware-spezifischen Kalkül basieren. Die Klassifizierung ergibt sich aus der hardware-orientierten Repräsentation der Schaltungen. Ein Benutzer kann durch interaktives Anwendung von Transformationen den Datenpfad einer Beschreibung auf Register-Transfer-Ebene ändern. Da die Umformungen nicht korrektheitserhaltend sind, wird nach ihrer Durchführung die Korrektheit durch spezielle Prüfprogramme überprüft. Auftretende Fehler werden automatisch korrigiert.
RLEXT berechnet eine Reihe von Forderungen, die erfüllt sein müssen, um die Korrektheit eines Entwurfs zu garantieren. Als problematisch erweist sich, daß die berechneten Forderungen lediglich axiomatisierte Plausibilitätskriterien dar-

stellen, von denen nicht formal erwiesen ist, daß sie zur Sicherstellung der Korrektheit hinreichend sind.

2.3.3 DDD

In *DDD (Digital Design Derivation)* [JM97, JB97] werden Schaltungen durch funktionale, nichttypisierte Lisp-Ausdrücke erster Ordnung dargestellt. Als Eingabesprache dient eine Erweiterung des Lisp-Dialekts Scheme. Neben einfachen Transformationen bietet DDD auch komplexere synthesetypische Transformationen an. Das Verhalten eines Entwurfs wird als endlicher Zustandsautomat bzw. als Transitionssystem spezifiziert. Durch sukzessives Verfeinern des Transitionssystems leitet das DDD-System eine Implementierung ab. Anzumerken ist, daß die zum Teil sehr aufwendigen Transformationen weder auf elementare Kalküle zurückgeführt wurden, noch ihre Korrektheit formal verifiziert wurde.
Da eine an Lisp angelehnte Notation gewählt worden ist, kann jeder Ausdruck sehr leicht in einem Lisp-Interpreter simuliert werden. Als problematisch erweist sich jedoch die fehlende Typisierung, die zu unrealistischen Schaltungsbeschreibungen führen kann. Obwohl jeder Ausdruck in DDD ein ausführbares Lisp-Programm ist, bedarf es Einschränkungen, die gewährleisten, daß realisierbare Hardwarekomponenten beschrieben werden.
In [BJ93] wird der DDD-FM9001 Mikroprozessor vorgestellt, der eine unter Verwendung des DDD-Systems abgeleitete Implementierung der FM9001 32-Bit Prozessorarchitektur darstellt. Als Spezifikation diente die in [BHK94] verifizierte Nqthm [BM88] Beschreibung. Mit *DRS (Derivational Reasoning System)* [Der95] steht DDD als kommerzielles Werkzeug zur Verfügung.

2.3.4 TRADES

Bei *TRADES (Transformational Design System)* [Mid94, Mid97] handelt es sich um ein interaktives transformationsbasiertes Synthesesystem. Eine VHDL-Beschreibung wird durch einen *SIL-Graphen (SPRITE Input Language)* repräsentiert. SIL-Graphen sind *Signalflußgraphen*. Im Unterschied zu *Kontrolldatenflußgraphen* modellieren Signalflußgraphen den Datenfluß über den Austausch von *Tokens*. Somit können weitere zeitliche Eigenschaften, wie z.B. Verzögerungen, repräsentiert werden, die über die partielle Ordnung, die sich aus den Datenabhängigkeiten ergibt, nicht modelliert werden können.
Die Transformationen werden auf den SIL-Graphen ausgeführt. Neben algebraischen Transformationen und Transformationen zur Verfeinerung stehen Umformungen wie z.B. Retiming oder Scheduling zur Verfügung, mit denen das Zeitverhalten und der Platzbedarf optimiert werden können. Die Korrektheit der in dem System verwendeten Transformationen wurde unter Verwendung eines

Theorembeweisers in [Raj95] verifiziert.
Mit TRADES wurde unter anderem ein Video-Signalprozessor entwickelt, dessen Güte mit einem von erfahrenen Designern durchgeführten Handentwurf vergleichbar war [MR96].

2.3.5 LAMBDA/DIALOG

LAMBDA/DIALOG [FFFH90] ist eines der wenigen kommerziellen Synthesewerkzeuge der formalen Synthese. Den Kern des Werkzeugs bildet LAMBDA, ein Theorembeweiser für Prädikatenlogik höherer Ordnung. LAMBDA unterstützt ausschließlich synchrone Schaltungen auf Gatterebene. DIALOG ist ein interaktives, graphisches Entwurfswerkzeug.
Die Spezifikation einer Schaltung wird als Formel höherwertiger Logik eingegeben. Es lassen sich Komponenten spezifizieren, die hierarchisch verfeinert werden können. Ein Schwerpunkt wird in LAMBDA auf die Sicherstellung der Konsistenz gelegt. Durch jeden Einfügeschritt kann es in der Schaltungsstruktur zu einem Kurzschluß oder kombinatorischen Zyklus kommen. Aus diesem Grund wird in DIALOG nach jeder Einfügeoperation die Schaltungsstruktur überprüft, wobei zunächst keine Einschränkungen bezüglich der zu synthetisierenden Spezifikation gemacht werden. Existiert zu einer Spezifikation keine Implementierung, stellt sich dies erst während des Syntheseablaufs heraus.
Es ist im allgemeinen nicht möglich, eine Spezifikation allein durch Anwenden von Einfügeregeln zu reduzieren, sondern es sind zusätzliche, interaktiv auszuführende logische Umformungsschritte erforderlich. Derartige Schritte erfordern logische Kenntnisse und Erfahrungen im Umgang mit dem Theorembeweiser LAMBDA. Das LAMBDA/DIALOG System ist erfolgreich bei der Synthese industrieller Designs eingesetzt worden [BBC+94, HM96].

2.3.6 VERITAS

VERITAS [HDL89, HD92] basiert auf einem Theorembeweiser für typisierte Prädikatenlogik höherer Ordnung. Der Theorembeweiser wurde um sogenannte *Techniken* erweitert, die es erlauben, Verfeinerungsschritte durchzuführen und aus einer vorgegebenen, verhaltensorientierten Schaltungsbeschreibung eine Implementierung abzuleiten. Eine Technik besteht aus einer *Teilzielfunktion* und einer *Validierungsfunktion*. Während die Teilzielfunktion das Beweisziel in Teilziele zerlegt, ergibt sich die Korrektheit des Verfeinerungsschrittes durch Anwenden der Validierungsfunktion. Somit gliedert sich die Synthese in zwei Phasen. In der ersten Phase wird die Zielbeschreibung sukzessive in mehrere Teile zerlegt. Aus den bewiesenen Teilzielen wird in der zweiten Phase durch Anwendung der Validierungsfunktion das Gesamtziel abgeleitet.

Dem Benutzer bietet VERITAS logische Elementarumformungen und einfache Schaltungstransformationen an. Da die Syntheseschritte auf der Ebene der elementaren, logischen Umformungen formuliert werden müssen, benötigt der Anwender einige Erfahrungen im Umgang mit dem Theorembeweiser.

2.3.7 HASH

HASH (Higher Order Logic Applied to Synthesis of Hardware) [EK95, BS99] verwendet den Theorembeweiser HOL [GM93] für Prädikatenlogik höherer Ordnung, um die Korrektheit der Synthese zu garantieren. Die Synthese wird in zwei Phasen aufgeteilt. In der ersten Phase wird der Entwurfsraum untersucht, um in der zweiten Phase die Schaltungstransformationen durchzuführen. Erst die zweite Phase sichert die Korrektheit, wogegen die erste Einfluß auf die Qualität des Ergebnisses hat.
Da die Korrektheit durch die zweite Phase garantiert wird, können in der ersten Phase beliebige Heuristiken oder Syntheseverfahren verwendet werden. Die gewonnenen Informationen leiten den Transformationsprozeß in der zweiten Phase. Sollte eine Heuristik fehlerhafte Ergebnisse liefern, wird dies, da die Synthese auf die Anwendung der verfügbaren logischen Transformationen eingeschränkt ist, entdeckt. Dabei ergibt sich das Problem, daß der Synthesealgorithmus innerhalb des Theorembeweisers rekonstruiert und formalisiert werden muß.
Aus einer Spezifikation auf der algorithmischen Ebene leitet HASH durch Anwenden logischer Transformationen in dem Theorembeweiser HOL eine strukturelle Implementierung auf der Register-Transfer-Ebene ab. Zu diesem Zweck wurde der Kern des Theorembeweisers HOL um weitere logische Transformationen erweitert. Kombinatorische Schaltungen werden als Funktionen, sequentielle als Mealy-Automaten repräsentiert. Durch Anwenden von Transformationen wird die Spezifikation in eine sogenannte *Single Loop Form (SLF)* überführt. Diese Darstellung ist zwar nicht eindeutig, doch alle von einer Beschreibung abgeleiteten SLF-Programme sind äquivalent. Unter Vorgabe eines Schnittstellenverhaltens wird einem SLF-Programm eine Implementierung zugeordnet.
HASH bietet eine Bibliothek von Synthesetransformationen wie z.B. Einplanung, Register-Allokation u.a. an, die automatisiert ausgeführt oder von dem Benutzer interaktiv ausgewählt werden können. Somit muß der Benutzer über ausreichende Kenntnisse im Umgang mit HOL verfügen. Hinzu kommt, daß als Beschreibungssprache eine Teilmenge der Logik des Theorembeweisers HOL dient.

2.3.8 Weitere Arbeiten

In vielen anderen Arbeiten werden transformative Ansätze vorgestellt. So bietet z.B. die *System Architect's Workbench* [TDW+88, WT89] Transformationen an,

um interaktiv den Entwurfsraum zu evaluieren. Die Transformationen erlauben es, eine Hierarchie einzuführen, Konstanten zu propagieren, Schleifen zu entrollen usw. Als Spezifikation dienen Verhaltensbeschreibungen in der *Instruction Set Processor Specification* Sprache. Intern werden diese Beschreibungen durch sogenannte *Value Trees* repräsentiert. In [MP83] wird die Korrektheit der Transformationen der System Architect's Workbench überprüft.
In [Cam89] wird der *Yorktown Silicon Compiler (YSC)* vorgestellt, welcher unter Anwendung korrektheitserhaltender Transformationen aus einem Schaltkreis, der im *Yorktown Internal Format* gegeben ist, eine Register-Transfer-Beschreibung ableitet. Insbesondere die Einplanung und Register-Allokation werden unterstützt. Die Korrektheit der Transformationen wird in [Cam89] gezeigt, wobei allerdings die Implementierung oder abgeleiteten Syntheseergebnisse nicht auf ihre Korrektheit überprüft werden.
In dem Synthesewerkzeug *HYPER* [CPR89, PR93] werden *Kontrolldatenflußgraphen* verwendet, um Verhaltensbeschreibungen in *SILAGE* zu repräsentieren. Das Werkzeug unterstützt insbesondere rechenintensive Echtzeitanwendungen. Durch die Anwendung von Transformationen werden Optimierungen ausgeführt und eine Register-Transfer-Beschreibung generiert. Die Transformationen sind jedoch nicht korrektheitserhaltend, da algebraische Transformationen existieren, durch deren Anwendung die Äquivalenz von Spezifikation und abgeleiteter Beschreibung nicht mehr erfüllt ist.
Das *CAMAD* [SP94, HP95] System verwendet Transformationen für die Einplanung und Register-Allokation. Eine Verhaltensbeschreibung in VHDL wird in ein *Extended Timed Petri Net (ETPN)* überführt. CAMAD partitioniert die ETPN-Repräsentation in einen Hard- und Softwareteil und führt die Hardware-Synthese durch, während die Software-Synthese von einem C-Compiler übernommen wird. Die Korrektheit der Transformationen wird nicht gezeigt.
Das interaktive Synthesewerkzeug *TRUST (Transformation-based User-directed Synthesis Tool)* [Har97] erzeugt durch Anwendung einer Folge korrektheitserhaltender Transformationen aus einer Verhaltensbeschreibung in *MINOLA* eine Implementierung auf der Register-Transfer-Ebene. In [Har97] wird zwar die Korrektheit der Transformationen nachgewiesen, deren programmtechnische Umsetzung bleibt aber ungeprüft.

2.4 Transformative Ansätze zur Verifikation

Transformative Ansätze können nicht nur im Bereich der formalen Synthese verfolgt werden, sondern können auch im Bereich der Post-Synthese-Verifikation zum Einsatz kommen. Die Äquivalenz von zwei Beschreibungen kann nachgewiesen werden, indem die Beschreibungen durch Anwenden einer Folge von formal korrekten Transformationen angeglichen werden.

In [FK91] wird z.B. unter Anwendung von Transformationen die Äquivalenz einer Spezifikation und einer abgeleiteten Register-Transfer-Implementierung gezeigt. Um dies sicherstellen zu können, ist die Synthese auf das Anwenden von Transformationen beschränkt. Die Register-Transfer-Beschreibung wird als *Flußgraph*, der durch symbolische Simulation gewonnen wird, dargestellt und durch Anwenden von Transformationen kanonisiert. Die Korrektheit des Syntheseergebnisses im Sinne einer *Pfadäquivalenz* ergibt sich, indem die ursprüngliche Spezifikation, die ebenfalls als Flußgraph gegeben ist, kanonisiert und mit dem Flußgraphen der Register-Transfer-Beschreibung verglichen wird.
Im Rahmen dieser Arbeit wird in Kapitel 6 ein Verfahren vorgestellt, daß die Äquivalenz zweier Beschreibungen nachweist, indem gezeigt wird, daß eine Folge von Transformationen existiert, die die eine in die andere Beschreibung umwandelt. Dabei ist es erforderlich, daß die beiden Beschreibungen *strukturelle Ähnlichkeiten* aufweisen. Das Verfahren wird aus diesem Grund verwendet, um die Korrektheit von Ergebnissen der Einplanungsphase zu verifizieren. Die Einplanung bzw. Ablaufplanung ist ein Syntheseschritt der High-Level-Synthese, in welchem die Operationen den Ausführungszeitpunkten zugeordnet werden. Insbesondere moderne Einplanungsverfahren wie z.B. *As Fast As Possible* [Cam91] arbeiten auf zyklischen Beschreibungen, so daß ihre Ergebnisse nicht durch Anwenden anderer Methoden der Post-Synthese-Verifikation, wie z.B. durch symbolische Simulation [REH99], überprüft werden können. Im Gegensatz zu [FK91] können die Resultate herkömmlicher Synthese- bzw. Einplanungsverfahren verifiziert werden.

2.5 Das TUD Transformationswerkzeug

Das *TUD Transformationswerkzeug (TUDT)* ist ein interaktives Werkzeug der formalen Synthese. Aus einer Menge allgemeingültiger, formal korrekter Transformationen kann ein Benutzer Transformationen auswählen und auf eine algorithmische Beschreibung anwenden. Neun Axiome und eine Regel ermöglichen das Umformen von Kontrollstrukturen. Sie sind korrektheitserhaltend im Sinne einer *Pfadäquivalenz*. Darüberhinaus stehen sechs Axiome zur Verfügung, um das Zeitverhalten einer Beschreibung zu verändern. Sie sind korrekt im Sinne einer *Berechnungsäquivalenz*. Da der hier vorgestellte Ansatz auf algorithmischen Beschreibungen arbeitet, stehen keine Transformationen wie z.B. Retiminig zur Verfügung, die die Register-Transfer-Struktur verändern. Kapitel 4 stellt die Äquivalenzbegriffe und Transformationen dar. Soll ein Transformationsprozeß automatisiert werden, können die Transformationen zu sogenannten *Meta-Transformationen* kombiniert werden.
Um einen formal korrekten Hardwareentwurf zu ermöglichen, ist es erforderlich, die Korrektheit der Implementierung nachzuweisen. Einige Werkzeuge der

formalen Synthese, wie z.B. DDD, verzichten darauf, die Korrektheit des Ergebnisses zu zeigen. Einige verifizieren dagegen nur Plausibilitätsforderungen wie z.B. RLEXT, während bei anderen die Transformationen, wie z.B. bei TRADES, formal verifiziert wurden, die Implementierung des Programms jedoch ungeprüft bleibt. In einigen Ansätzen, wie z.B. bei VERITAS, HASH oder LAMBDA/DIALOG, wird die Synthese in einen Theorembeweiser eingebettet. Die Verwendung eines Theorembeweisers stellt die Korrektheit sicher, setzt jedoch Kenntnisse und Erfahrungen im Umgang mit einem solchen Werkzeug voraus.
Die Korrektheit eines Ergebnisses des *TUD Transformationswerkzeugs (TUDT)* wird nachgewiesen, indem eine Post-Synthese-Verifikation durchgeführt wird. Ist es nach der Durchführung einer Folge von Transformationen nicht möglich, die Korrektheit zu zeigen, kann jeder Transformationsschritt einzeln verifiziert werden. Wie Abschnitt 5.5 zu entnehmen ist, konnten durch die Verifikation Fehler der Implementierung des Werkzeugs entdeckt und behoben werden.
Als Beispiel wird ein Verfahren zur automatisierten Einplanung von Pipelines vorgestellt (siehe Kapitel 5). Ausgehend von der Beschreibung eines sequentiellen Prozessors wird durch sukzessives Anwenden von Transformationen eine Implementierung als Pipeline abgeleitet. Treten in der Pipeline Konflikte durch Datenabhängigkeiten von Befehlen oder durch Beschränkungen der vorgegebenen Ressourcen auf, werden diese beispielsweise durch *Forwarding* oder durch das Einführen von *Pipeline-Registern* gelöst. Das Verfahren wurde unter anderem angewendet, um eine DLX-Pipeline [HP96] einzuplanen. Es zeigt sich, daß das automatisch eingeplante Ergebnis der klassischen, manuell erzeugten Lösung entspricht.
Im Unterschied zu den übrigen Werkzeugen der formalen Synthese kann das *TUD Transformationswerkzeug (TUDT)* auch zur Post-Synthese-Verifikation eingesetzt werden. Durch Anwenden von Transformationen kann die Äquivalenz zweier Beschreibungen gezeigt werden. Existiert eine Folge korrektheitserhaltender Transformationen, die die eine Beschreibung in die andere überführt, sind die beiden Beschreibungen äquivalent. Kapitel 6 stellt diesen Ansatz dar.
Als Beschreibungssprache dient die *Language of Labelled Segments (LLS)*, die in Kapitel 3 vorgestellt wird. Es handelt sich dabei um eine axiomatisierte Hardwarebeschreibungssprache, die es erlaubt, synchrone Transitionssysteme zu beschreiben. LLS ist zwar nicht so mächtig wie z.B. VHDL, besitzt aber eine formale Semantik, so daß diese Beschreibungssprache als Eingabesprache sowohl für Synthese- als auch für Verifikationswerkzeuge geeignet ist.

2.6 Einordnung in den Entwurfsfluß

Das Gebiet der High-Level-Synthese beschäftigt sich mit automatisierbaren Verfahren, die aus einer vorgegebenen algorithmischen Register-Transfer- eine struk-

turelle Beschreibung erzeugen. Die Synthese wird traditionell in eine Reihe von Schritten [Mic94] unterteilt.

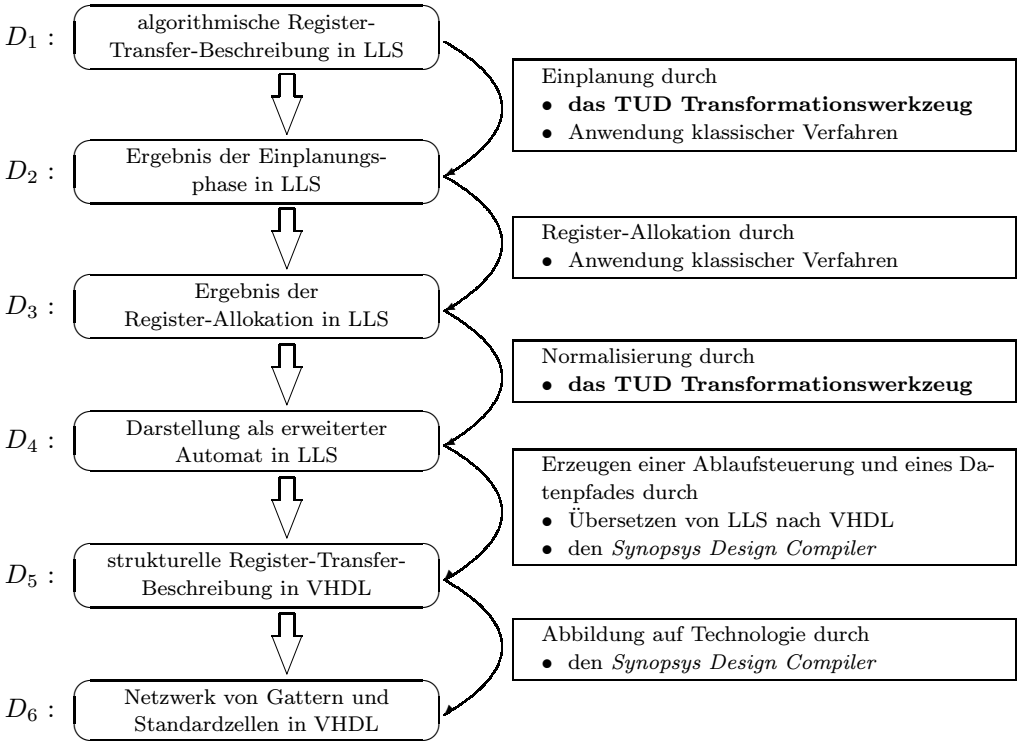


Abb. 2.2: Entwurfsschritte

Abbildung 2.2 gibt ein Beispiel für einen möglichen Entwurfsfluß. Ausgehend von einer algorithmischen Register-Transfer-Beschreibung D_1 wird, mehrere Syntheseschritte ($D_i \Rightarrow D_j$) durchlaufend, eine strukturelle Beschreibung D_6, die als Netzliste gegeben ist, abgeleitet.
Bei der *Einplanung* ($D_1 \Rightarrow D_2$) werden die Operationen der ursprünglich gegebenen Beschreibung auf Ausführungszeitpunkte festgelegt. Das in dieser Arbeit vorgestellte *TUD Transformationswerkzeug (TUDT)* arbeitet interaktiv, so daß ein Benutzer durch Anwenden von korrektheitserhaltenden Transformationen eine Einplanung vornehmen kann. Durch *Meta-Transformationen* können einzelne Transformationen kombiniert werden, so daß sich der Transformationsprozeß automatisieren läßt. Als Beschreibungssprache dient die experimentelle Hardwarebeschreibungssprache *Language of Labelled Segments (LLS)* (siehe Kapitel 3).
Den Variablen der ursprünglichen Beschreibung ordnet die *Register-Allokation* ($D_2 \Rightarrow D_3$) physikalisch aufzubauende Register zu. Ein Werkzeug zur Register-Allokation existiert noch nicht.
Algorithmische Beschreibungen können sequentielle Strukturen enthalten. Im nächsten Schritt $D_3 \Rightarrow D_4$ wird durch Normalisieren der LLS-Beschreibung (siehe Abschnitt 3.3) ein erweiterter endlicher Automat erzeugt. Das Normalisieren

wird, um die Korrektheit der Umformung gewährleisten zu können, auf eine Folge von Transformationen zurückgeführt, die durch das *TUD Transformationswerkzeug (TUDT)* ausgeführt werden.
Um kommerzielle Synthesewerkzeuge wie z.B. den *Synopsys Design Compiler* [Syn98a] zu nutzen, wird die LLS-Beschreibung nach VHDL übersetzt (siehe Abschnitt 3.5). Eine Ablaufsteuerung und der Datenpfad werden von dem *Synopsys Design Compiler* in dem Schritt $D_4 \Rightarrow D_5$ generiert und durch $D_5 \Rightarrow D_6$ auf eine Technologie abgebildet.

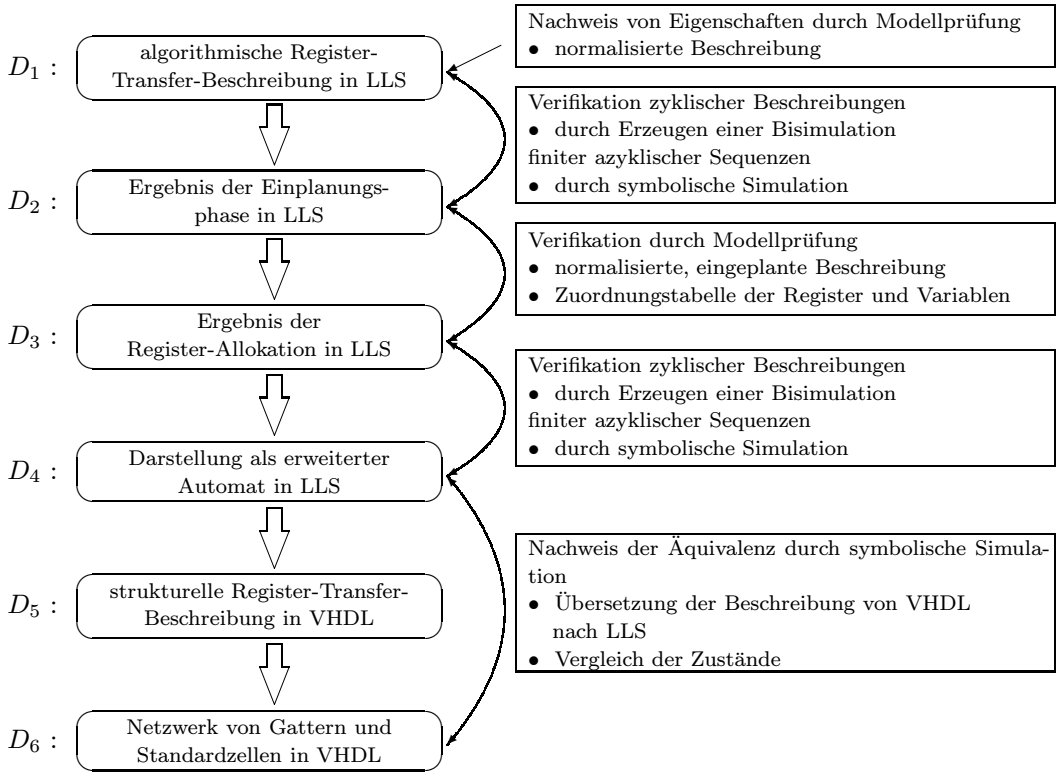


Abb. 2.3: Verifikation der Entwurfsschritte

Da die Syntheseverfahren und somit auch die Synthesewerkzeuge immer komplexer werden, gewinnt der Nachweis, daß ein Syntheseergebnis hinsichtlich der Spezifikation korrekt ist ($D_i \Leftrightarrow D_j$), an Bedeutung. Während es für die Logik-Verifikation ($D_5 \Leftrightarrow D_6$) bereits kommerzielle Werkzeuge gibt, fehlen vergleichbar erfolgreiche Werkzeuge zur Überprüfung der restlichen Schritte. Der in Abbildung 2.3 dargestellte Entwurfsfluß unterstützt die Verifikation der verbleibenden Syntheseschritte.
So läßt sich die Einplanung durch symbolische Simulation [REH99] oder Erzeugen einer Bisimulation (siehe Kapitel 6) verifizieren ($D_1 \Leftrightarrow D_2$). Die symbolische Simulation überprüft die Berechnungsäquivalenz von finiten azyklischen Beschreibungsteilen, wogegen der Nachweis einer Bisimulation die Äquivalenz zyklischer Beschreibungen zeigt. Da die zur Erzeugung der Bisimulation verwen-

dete Technik *strukturelle Ähnlichkeiten* voraussetzt, kann es notwendig werden, die beiden Verfahren zu kombinieren. Entsprechend wird das korrekte Erzeugen des erweiterten endlichen Automats überprüft ($D_3 \Leftrightarrow D_4$). Das *TUD Transformationswerkzeug (TUDT)* ist zwar ein Werkzeug der formalen Synthese, kann aber auch zur Verifikation der Schritte $D_1 \Leftrightarrow D_2$ und $D_3 \Leftrightarrow D_4$ eingesetzt werden, indem durch Anwenden von Transformationen eine Bisimulationsrelation aufgestellt wird.
Die Register-Allokation kann durch eine Modellprüfung verifiziert werden ($D_2 \Leftrightarrow D_3$) [BRHE00, ABRM98]. Aus der ursprünglichen Beschreibung wird ein Transitionssystem extrahiert, das die Verwendung der Variablen als Quelle oder Ziel von Anweisungen kodiert. Anhand der Zuordnungstabelle von Variablen und Registern wird die Unvereinbarkeit der Lebenszeiten überprüft. Ein Werkzeug zur Verifikation der Register-Allokation ist Gegenstand der derzeitigen Forschung.
Der *Synopsys Design Compiler* erzeugt die strukturelle Repräsentation und bildet diese auf eine Technologie ab. Aus diesem Grund wird überprüft, ob die als Netzliste gegebene, strukturelle Beschreibung und die Darstellung als erweiterter endlicher Automat äquivalent sind ($D_4 \Leftrightarrow D_6$). Der symbolische Simulator wird zur Zeit derart erweitert, daß auch die Äquivalenz einer Netzliste und einer Verhaltensbeschreibung gezeigt werden kann. Da das Syntheseergebnis als VHDL-Beschreibung gegeben ist, wird die Beschreibung zur Äquivalenzprüfung nach LLS zurückübersetzt. Die Übersetzung von VHDL nach LLS ist auf Netzlisten beschränkt.
Die erwähnten Verifikationsschritte sichern, daß die Spezifikation und das Syntheseergebnis äquivalent sind. Mit diesen Methoden läßt sich jedoch nicht überprüfen, ob die Spezifikation gewissen Eigenschaften genügt. Durch Modellprüfung können Eigenschaften wie z.B. der Lebendigkeit oder Sicherheitseigenschaften verifiziert werden. Es wird gezeigt, daß die Spezifikation D_1 jene geforderten Eigenschaften besitzt. Alle übrigen Beschreibungen sind ungleich komplexer und sind aus der Spezifikation abgeleitet. Durch Normalisieren der Spezifikation wird ein Transitionssystem gewonnen. Ein Werkzeug, das die zugehörige Transitionsrelation aufbaut, wird derzeit entwickelt, während ein Modellprüfer bereits existiert.

2.7 Zusammenfassung

Das *TUD Transformationswerkzeug (TUDT)* ist ein interaktives Werkzeug der formalen Synthese. Die Synthese wird nicht in einen Theorembeweiser eingebettet, da der Umgang mit dem Beweissystem große Erfahrung voraussetzt. Die Korrektheit der Synthese wird durch eine Post-Synthese-Verifikation sichergestellt. Die Äquivalenz von Spezifikation und Implementierung wird durch symbolische Simulation nachgewiesen. Das Transformationswerkzeug kann nicht nur zur Synthese, sondern auch zur Verifikation eingesetzt werden.

Im Entwurfsfluß wird das Werkzeug verwendet, um eine Einplanung vorzunehmen. Werden andere Verfahren angewendet, so kann das Werkzeug die Korrektheit des Ergebnisses der Einplanung überprüfen. Darüberhinaus kann mit Hilfe von Transformationen eine Beschreibung normalisiert werden. Ein Nachweis der Korrektheit ist ebenfalls möglich.

Kapitel 3

Algorithmische Schaltungsbeschreibungen

3.1 Wahl einer Beschreibungssprache

Der Entwurf digitaler Systeme setzt sich aus mehreren Teilschritten auf verschiedenen Abstraktionsebenen zusammen. Für jede Ebene existieren verschiedene Repräsentationsformen. In dieser Arbeit werden Systeme auf der algorithmischen und der Register-Transfer-Ebene dargestellt. Zur Repräsentation bietet sich eine universelle Sprache wie z.B. VHDL an. VHDL ist hauptsächlich als Simulationssprache entworfen worden und enthält daher viele Konstrukte zur Beschreibung verschiedener Zeitmodelle und physikalischer Eigenschaften. Es gibt eine Vielzahl von Möglichkeiten, Systeme in VHDL zu repräsentieren. Je nach Art der Darstellung werden unterschiedliche Syntheseergebnisse erzielt [PRK+96]. VHDL ist somit für die Synthese sowie für die Verifikation nur bedingt geeignet. Dies führt dazu, daß Entwurfswerkzeuge nur Teilmengen von VHDL als Eingabe erlauben oder einen bestimmten Beschreibungsstil voraussetzen. Beispiele sind in [MR96] oder [EKM96] zu finden. Internationale Standards existieren zur Zeit nicht.
Schwerer wiegt jedoch, daß VHDL keine formale Semantik besitzt. Um Beschreibungen formal verifizieren zu können, ist es notwendig, daß die Beschreibungssprache, die zur Repräsentation verwendet wird, eine formal definierte Semantik besitzt. [MP83] stellt z.B. einen der ersten Ansätze dar, die Bedeutung einer Hardwarebeschreibungssprache zu formalisieren. In [MP83] wird ein auf regulären Ausdrücken basierendes Modell zur Beschreibung der Interaktionen eines Schaltkreises mit seiner Umgebung entwickelt, mit dessen Hilfe eine Verhaltensäquivalenz von zwei Beschreibungen nachgewiesen werden kann.
Mitte der 80er Jahre wurde die Hardwarebeschreibungssprache *SMAX (small and axiomized)* [Eve91] entwickelt, die eine formale Semantik besitzt. Die Axiomatisierung von SMAX besteht aus einer Reihe von Regeln, die bestimmen, wie

die Konstrukte der Sprache in Formeln der Prädikatenlogik zu transformieren sind. Abbildung 3.1 stellt einen SMAX-Ausdruck und die daraus resultierenden Regeln der Prädikatenlogik dar. Die Funktion $\mathtt{x}(t)$ gibt an, welchen Wert die Variable `x` zu dem Zeitpunkt t annimmt. Für die Synthese ist SMAX jedoch nur unzureichend geeignet, da z.B. indirekte Zugriffe auf Felder nicht möglich sind.

```
IF p
  THEN x←y-1
  ELSE x←y
```

$$\forall t:\quad (t>0)\wedge(\mathtt{p}(t-1)=1)\Longrightarrow(\mathtt{x}(t)=\mathtt{y}(t-1)-1)$$
$$(t>0)\wedge(\mathtt{p}(t-1)=0)\Longrightarrow(\mathtt{x}(t)=\mathtt{y}(t-1))$$

Abb. 3.1: Regeln der Prädikatenlogik für einen SMAX-Ausdruck

Aus diesem Grund wurde, ausgehend von SMAX, im Rahmen dieser Arbeit die *Language of Labelled Segments (LLS)* [EHR98, Hin98b] entwickelt. Es handelt sich dabei um eine experimentelle, axiomatisierte Hardwarebeschreibungssprache, die es erlaubt, synchrone Transitionssysteme [MP91] auf der algorithmischen und der Register-Transfer-Ebene zu beschreiben. Nebenläufige Prozesse können nicht dargestellt werden. Die Semantik einer LLS Beschreibung ist durch einen erweiterten endlichen Automaten gegeben. Wie das Beispiel in Abbildung 3.2 zeigt, können an den Kanten des Zustandsdiagramms boolesche Ausdrücke als Bedingungen für einen Übergang und auszuführende Datenoperationen notiert werden.

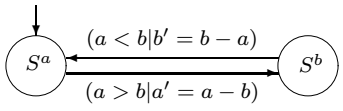


Abb. 3.2: Ein erweiterter endlicher Automat zur Bestimmung des GGT

Eine LLS-Beschreibung besteht aus einer Menge sogenannter Segmente, die jeweils einen bestimmten Systemzustand repräsentieren. Durch Markierungen der Segmente ist der Kontrollfluß festgelegt.
In den folgenden Abschnitten werden das Grundkonzept von LLS erläutert und eine Normalform definiert. Abschnitt 3.4 gibt die Semantik einer LLS-Beschreibung an. Die Möglichkeiten, LLS nach VHDL zu übersetzen, so daß auch kommerzielle Synthesewerkzeuge zum Einsatz kommen können, werden im Anschluß daran erläutert. Eine Zusammenfassung beschließt das Kapitel.

3.2 Language of labelled Segments

Die experimentelle Hardwarebeschreibungssprache LLS wird als Eingabesprache für Synthese- und Verifikationswerkzeuge verwendet. Eine LLS-Beschreibung repräsentiert ein deterministisches, synchrones Transitionssystem. Eine Beschreibung besteht aus einer Menge von *Segmenten* der Form:

<Marke>: <Segmentkörper>;

Der Körper jedes Segments enthält parallele oder sequentielle Anweisungen. Da ein Segment einem Zustand eines Systems bzw. eines Automaten entspricht, führt jedes Segment mindestens eine Anweisung aus. Klammern umschließen die von Kommas "," getrennten, gleichzeitig auszuführenden Anweisungen. Parallele Anweisungen können innerhalb eines Zeitschrittes ausgeführt werden. Ein Semikolon ";" trennt sequentielle Anweisungen, deren Ausführung mehrere Zeitschritte benötigt. Zuweisungen werden mit Hilfe von ← angegeben. *If-then-else*-Strukturen realisieren Verzweigungen. Schleifen müssen über bedingte Sprünge modelliert werden, explizite Schleifenkonstrukte sind in LLS nicht definiert, so daß der Kontrollfluß innerhalb eines Segments azyklisch ist und die Berechnung, die ein Segment ausführt, immer terminiert.

Beispiel 3.1 *Die parallelen Anweisungen* `(a←b,b←a)` *tauschen die Inhalte der Variablen* `a` *und* `b` *aus. Eine zusätzliche Variable wird nicht benötigt, da der Wert einer Zuweisung sich durch die Variablenwerte bestimmt, die einen Zeitschritt zuvor gültig sind. Auch if-then-else-Anweisungen können parallel, z.B. zu Zuweisungen, ausgeführt werden:* `(a←b,if p then c←d else e←f)`*. Der Buchstabe* `p` *symbolisiert eine boolesche Bedingung. Es ist aber auch möglich, die folgenden, äquivalenten Anweisungen zu spezifizieren:* `if p then (a←b,c←d); else (a←b,e←f);` *. Jedes der Beispiele wird innerhalb eines Zeitschrittes ausgeführt.*
Die Zuweisungen `(a←b);(c←a)` *werden sequentiell realisiert. Dies benötigt zwei Zeitschritte. Nach der Ausführung speichern sowohl* `a`*,* `b` *als auch* `c` *einen identischen Wert.*

Die Werte der Quellvariablen in LLS-Ausdrücken werden durch den vorangehenden Zeitschritt bestimmt.

Beispiel 3.2 *Abbildung 3.3 stellt zwei parallele Zuweisungen dar. Die Bedeutung der Struktur ist durch die Regeln der Prädikatenlogik gegeben. Die Funktion* $\mathtt{a}(t)$ *gibt an, welchen Wert die Variable* `a` *zu dem Zeitpunkt* t *annimmt. Die Variablen* `a` *und* `b` *tauschen ihre Werte aus, da der Wert der Quellvariablen durch den vorhergehenden Zeitschritt bestimmt wird.*

`(a←b,b←a);` $\qquad\qquad \forall t \,:\, (t > 0) \Longrightarrow (\mathtt{a}(t) = \mathtt{b}(t-1) \wedge \mathtt{b}(t) = \mathtt{a}(t-1))$

Abb. 3.3: Bedeutung eines parallelen LLS-Ausdrucks

Abbildung 3.4 stellt die Bedeutung eines sequentiellen Ausdrucks anhand von Regeln der Prädikatenlogik dar. Werden die Anweisungen ausgeführt, wird den Variablen `a` *und auch* `c` *der Wert von* `b` *zugewiesen.*

(a←b);(c←a); $\qquad \forall t \,:\, (t > 0) \Longrightarrow (\mathtt{a}(t) = \mathtt{b}(t-1) \wedge \mathtt{c}(t+1) = \mathtt{a}(t))$

Abb. 3.4: Bedeutung eines sequentiellen LLS-Ausdrucks

Marken bestimmen den Kontrollfluß einer LLS-Beschreibung. Jedes Segment besitzt eine eindeutige *Eingangsmarke*, die das Segment eindeutig kennzeichnet, und eine Menge von *Ausgangsmarken*, die angeben, welches Segment im nächsten Schritt zu bearbeiten ist. Zur Vereinfachung wird im folgenden die Eingangsmarke eines Segments oft als *Marke* bezeichnet. Da Verzweigungen möglich sind, kann ein Segment mehrere verschiedene Ausgangsmarken besitzen. Am Ende jedes Pfades durch den Segmentkörper findet sich genau eine Ausgangsmarke. Eine Beschreibung ist durch eine Menge von Segmenten und der Nennung einer *Anfangsmarke*, der Eingangsmarke des Segments, mit deren Ausführung die Berechnung begonnen wird, gegeben.

Beispiel 3.3 *In Abbildung 3.5(a) bestimmt die Beschreibung $\mathcal{B}$ den größten gemeinsamen Teiler zweier Zahlen* a *und* b*. Die Beschreibung besteht aus einem Segment S^a, das zwei sequentielle if-then-else-Anweisungen ausführt. Durch die Ausgangsmarke S^a wird der Kontrollfluß derart geleitet, daß das Segment erneut ausgeführt wird, so daß eine Schleife beschrieben wird. Der Kontrollfluß innerhalb des Segments ist jedoch azyklisch.*
Die Beschreibung $\mathcal{B}'$ aus 3.5(b) besteht aus zwei Segmenten S^a und S^b. Im Unterschied zu 3.5(a) benötigt es nur einen Zeitschritt, um S^a bzw. S^b auszuführen. Die Ausgangsmarke S^b des Segments S^a verweist auf das Segment S^b und umgekehrt. Auch diese Beschreibung beschreibt somit eine Schleife.

a) $\mathcal{B} = (\{S^a\}, S^a, \{a, b, c\})$ mit dem Segment

```
S^a :
  IF a>b
    THEN a←a-b;
    ELSE c←a;
  IF a<b
    THEN b←b-a;
    ELSE c←b;
  S^a;
```

b) $\mathcal{B}' = (\{S^a, S^b\}, S^a, \{a, b, c\})$ mit den Segmenten

```
S^a :
  IF a>b
    THEN a←a-b;
         S^b;
    ELSE c←a;
         S^b;
S^b :
  IF a<b
    THEN b←b-a;
         S^a;
    ELSE c←b;
         S^a;
```

Abb. 3.5: GGT als Beispiel für eine LLS-Beschreibung

Formal ist eine LLS-Beschreibung als Dreitupel definiert.

Definition 3.1 *Eine LLS-Beschreibung $\mathcal{B}$ ist durch $\mathcal{B} = (\mathcal{S}_\mathcal{B}, S^{init}, \mathcal{V}_\mathcal{B})$ mit*

- *$\mathcal{S}_\mathcal{B}$ als Menge der Segmente,*

- $S^{init} \in \mathcal{S}_\mathcal{B}$ *als Anfangsmarke und*
- $\mathcal{V}_\mathcal{B} = \{v_1, ..., v_n\}$ *als geordnete Menge der Variablen*

gegeben.

Da die Definition einer Anfangsmarke und die Angabe von Ausgangsmarken innerhalb der Segmentkörper eine eindeutige Ausführung der Beschreibung gewährleisten, ist die Reihenfolge der Segmente beliebig. Die Menge der Variablen $\mathcal{V}_\mathcal{B} = \{v_1, ..., v_n\}$ wird als geordnet angenommen, um die Darstellung von Berechnungen in Kapitel 4 zu vereinheitlichen und zu vereinfachen.

Beispiel 3.4 *In Abbildung 3.5(b) ist die Beschreibung $\mathcal{B}'$ durch $\mathcal{B}' = (\{S^a, S^b\}, S^a, \{a, b, c\})$ gegeben. Die Beschreibung besteht aus den Segmenten S^a und S^b. Begonnen wird die Berechnung mit dem Segment S^a, da diese Marke als Anfangsmarke angegeben wird. Die verwendeten Variablen sind* `a`, `b` *und* `c`.

Das Zeitkonzept von LLS schließt Nichtdeterminismus und Nebenläufigkeit bei der Berechnung der Anweisungen aus.

3.3 Normalform einer LLS-Beschreibung

In LLS-Beschreibungen können Segmente auftreten, die sequentielle Anweisungen enthalten und somit mehr als nur einen Zeitschritt zur Ausführung benötigen. Ist in einer Beschreibung kein solches Segment zu finden, ist die Beschreibung in *Normalform* bzw. kann die Beschreibung als erweiterter endlicher Automat dargestellt werden. Ein Segment entspricht nämlich genau dann einem Zustand eines Automaten, wenn es einen Zeitschritt zur Ausführung benötigt.

Definition 3.2 (Normalform)
Eine LLS-Beschreibung $\mathcal{B}$ ist in Normalform, wenn jedes Segment $S^a \in \mathcal{S}_\mathcal{B}$ genau einen Zeitschritt zur Ausführung benötigt.

Weiterhin gilt:

Satz 3.1 *Jede LLS-Beschreibung $\mathcal{B}$ läßt sich normalisieren.*

Der Algorithmus 3.1 *Normalisiere(S_1^x; ...; S_n^x;)* erlaubt es, einzelne Segmente zu normalisieren. S_1^x bis S_n^x repräsentieren die Anweisungen des Segmentkörpers. Handelt es sich bei S_i^x um eine if-then-else-Anweisung, bezeichnet ${S_i^x}^T$ den then- und ${S_i^x}^E$ den else-Zweig. Durch Normalisieren jedes Segments einer Beschreibung läßt sich eine Beschreibung in die Normalform überführen. Die Umwandlung erfolgt unter Anwendung formaler Transformationen (siehe Abschnitt 4.6), die die

Pfadäquivalenz (siehe Abschnitt 4.3) der ursprünglichen und der normalisierten Beschreibung erhalten.

Algorithmus 3.1 Normalisiere

input *eine Folge von sequentiellen Anweisungen S_1^x bis S_n^x eines Segments S^x*

1. **if** $(n = 1) \wedge (S_1^x$ *ist eine if-then-else-Anweisung*$)$ **then**
2. *Normalisiere($S_1^{x\ T}$);*
3. *Normalisiere($S_1^{x\ E}$);*
4. **return** S_1^x*;*
5. **elsif** $(n = 2) \wedge (S_2^x$ *ist eine Ausgangsmarke*$)$ **then**
6. **if** *(S_1^x ist eine if-then-else-Anweisung)* **then**
7. *Ziehe die Ausgangsmarke S_2^x in die if-then-else-Anweisung S_1^x hinein* **[A6]***;*
8. *Normalisiere($S_1^{x\ T}$);*
9. *Normalisiere($S_1^{x\ E}$);*
10. **return** S_1^x*;*
11. **else**
12. **return** $S_1^x; S_2^x$*;*
13. **fi***;*
14. **elsif** $(n > 1)$ **then**
15. *Erzeuge ein neues Segment S^{neu}, das die sequentiellen Anweisungen S_2^x bis S_n^x aufnimmt* **[R1]***;*
16. *Normalisiere($S^{neu} : S_2^x; ...; S_n^x;$);*
17. **return** *Normalisiere($S_1^x; S^{neu};$);*
18. **else**
19. **return** S_1^x*;*
20. **fi***;*

Segmente können beliebig viele und verschachtelte sequentielle Anweisungen umfassen. Die Funktion *Normalisiere* verkürzt eine Folge von sequentiellen Anweisungen durch Erzeugen eines neuen Segments S^{neu} und durch Verschieben von Anweisungen nach S^{neu}, so daß eine Anweisung zurückbleibt (Zeilen 14-17). Abbildung 3.6 zeigt an einem Beispiel mit drei sequentiellen Anweisungen das Erzeugen eines neuen Segments durch das Anwenden der Regel R1 (siehe Abschnitt 4.6). In Zeile 16 wird S^{neu} rekursiv normalisiert.

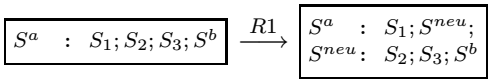


Abb. 3.6: Erzeugen eines neuen Segments

Besteht die Folge der sequentiellen Anweisungen aus einer Verzweigung (Zeilen 1-4) oder aus einer Verzweigung gefolgt von einer Ausgangsmarke (Zeile 6-10), werden sowohl der then- (Zeilen 2, 8) also auch der else-Zweig (Zeilen 3, 9) rekursiv normalisiert. Da eine Verzweigung ebenfalls sequentielle Anweisungen enthalten kann und somit die Notwendigkeit besteht, Anweisungen in neue Segmente zu verschieben, muß im letzteren Fall durch Anwenden des Axioms A6 (siehe Abschnitt 4.6) die Ausgangsmarke in die if-then-else-Anweisung hineingezogen werden (Zeile 7). Abbildung 3.7 gibt ein Beispiel.

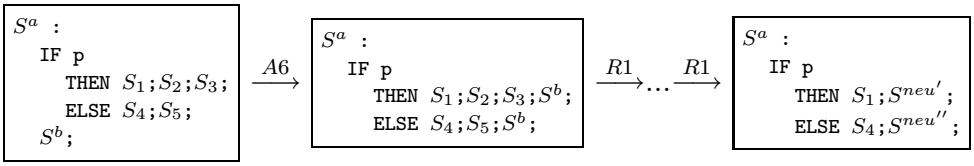


Abb. 3.7: Hineinziehen einer Ausgangsmarke in eine if-then-else-Anweisung

Wird Zeile 4, 10, 12 oder 19 erreicht, ist $n = 1$, so daß die Rekursion abgebrochen werden kann.

Beispiel 3.5 *Die Beschreibung $\mathcal{B}$ in Abbildung 3.5(a) berechnet für zwei Zahlen* a *und* b *den größten gemeinsamen Teiler. Durch Anwenden des Algorithmus 3.1 wird die Beschreibung normalisiert. Zeile 15 führt ein neues Segment S^b ein. Bevor S^a in die Normalform überführt wird (Zeile 17), normalisiert Zeile 16 S^b. Da S^b eine Verzweigung gefolgt von einer Ausgangsmarke enthält, wird die Ausgangsmarke in die Verzweigung hineingezogen (Zeile 7). Durch den erneuten, rekursiven Aufruf werden der then- (Zeile 8) und der else-Zweig (Zeile 9) der if-then-else-Anweisung normalisiert. Weil in beiden Zweigen eine Zuweisung gefolgt von einer Ausgangsmarke ausgeführt wird, sind keine weiteren Transformationen erforderlich (Zeile 12). Im Anschluß wird mit dem Segment S^a analog verfahren (Zeile 17). Das Ergebnis $\mathcal{B}'$ ist in 3.5(b) dargestellt.*

Die Normalform ist im allgemeinen nicht kanonisch. Werden z.B. zwei Beschreibungen normalisiert, deren Segmente *berechnungsäquivalent* (siehe Abschnitt 4.4) sind, so können sich unterschiedliche Beschreibungen ergeben. Die *Berechnungsäquivalenz* zweier Segmente sagt aus, daß diese Segmente die gleichen Werte bei gleicher Initialisierung berechnen, wobei jedoch die zur Berechnung benötigten Zeiten voneinander abweichen können. Diese Differenzen im Zeitverhalten sind der Grund für Unterschiede der normalisierten Beschreibungen. Die Eigenschaft der *Berechnungsäquivalenz* zweier Segmente kann durch das Normalisieren verloren gehen.

Beispiel 3.6 *Die Beschreibung $\mathcal{B}''$ aus Abbildung 3.8 ist berechnungsäquivalent zu $\mathcal{B}$ aus Abbildung 3.5(a). S^a aus $\mathcal{B}''$ benötigt einen Zeitschritt, S^a aus $\mathcal{B}$ führt die Berechnungen in zwei Zeitschritten aus. Nach Ausführung der Segmente ergeben sich die gleichen Werte für die Variablen* a, b *und* c*, so daß die Segmente*

S^a aus $\mathcal{B}$ und S^a aus $\mathcal{B}''$ berechnungsäquivalent sind.
Im Unterschied zu $\mathcal{B}$ ist $\mathcal{B}''$ bereits normalisiert. Die Normalform $\mathcal{B}'$ von $\mathcal{B}$ ist in Abbildung 3.5(b) dargestellt. Es existiert zu keinem Segment aus $\mathcal{B}'$ ein berechnungsäquivalentes Segment in $\mathcal{B}''$.

$\mathcal{B}'' = (\{S^a\}, S^a, \{a, b, c\})$ mit dem Segment

```
S^a :
  (IF a>b
    THEN a←a-b,
  ELSIF a<b
    THEN b←b-a,
  ELSE   c←a);
  S^a;
```

Abb. 3.8: Eine berechnungsäquivalente Beschreibung zur GGT-Berechnung

Obwohl die Normalformen *pfadäquivalenter* Beschreibungen (siehe Abschnitt 4.3) im allgemeinen nicht identisch sind, bleibt die *Pfadäquivalenz* erhalten. Zwei Beschreibungen sind *pfadäquivalent*, wenn die Werte der Variablen in jedem Schritt ihrer Berechnungen übereinstimmen. Die normalisierten Beschreibungen unterscheiden sich, da einerseits die parallelen Anweisungen permutiert und andererseits durch negierte Bedingungen der Verzweigungen Pfade in den Beschreibungen vertauscht sein können.

a) $\mathcal{B}'''$ = $(\{S^a\}, S^a, \{a, b, c\})$ mit dem Segment

```
S^a :
  IF ¬(a>b)
    THEN c←a;
    ELSE a←a-b;
  IF ¬(a<b)
    THEN c←b;
    ELSE b←b-a;
  S^a;
```

b) $\mathcal{B}'''^{NF}$ = $(\{S^a, S^b\}, S^a, \{a, b, c\})$ mit den Segmenten

```
S^a :
  IF ¬(a>b)
    THEN c←a;
         S^b;
    ELSE a←a-b;
         S^b;
S^b :
  IF ¬(a<b)
    THEN c←b;
         S^a;
    ELSE b←b-a;
         S^a;
```

Abb. 3.9: Eine pfadäquivalente Beschreibung zur GGT-Berechnung

Beispiel 3.7 *In Abbildung 3.9 sind eine zu $\mathcal{B}$ aus Abbildung 3.5(a) pfadäquivalente Beschreibung $\mathcal{B}'''$ und ihre Normalform $\mathcal{B}'''^{NF}$ dargestellt. Die Beschreibungen unterscheiden sich durch die invertierten Bedingungen der Verzweigungen* `¬(a>b)` *bzw.* `¬(a<b)`*. Die Pfadäquivalenz der Beschreibungen bleibt erhalten.*

3.4 Die Semantik einer LLS-Beschreibung

Ein Transitionssystem ist nach Manna und Pnueli [MP91] durch (Π, Σ, Φ, s_1) gegeben mit:

- $\Pi = \{\pi, v_1, ..., v_n\}$ der Menge der Variablen,
- $\Sigma = \{s_1, ..., s_m\}$ der Menge der Zustände,
- $\Phi = \{\tau_1, ..., \tau_k\}$ der Menge der Zustandsübergangs- oder Transitionsfunktionen und
- s_1 als Anfangszustand.

Die Menge der Variablen enthält einerseits Datenvariablen v_1 bis v_n beliebigen Typs und andererseits die Variable π, die den Kontrollfluß steuert. Ein Zustand des Systems versteht sich als Interpretation der Menge der Variablen Π, d.h. ein Zustand wird durch eine bestimmte Kombination der Variablenwerte charakterisiert.
Eine Berechnung des Systems ist durch eine Folge von Zustandsübergängen gegeben. Die Transitionsfunktion $\tau : \Sigma \rightarrow \Sigma$ bestimmt für einen Zustand den Folgezustand:

$$s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} ... \xrightarrow{\tau_i} s_l \xrightarrow{\tau_j} ...$$

Jeder Transition τ ist eine Transitionsrelation $\rho_\tau(\Pi, \Pi')$ zugeordnet. Die Transitionsrelation setzt die Werte der Variablen eines Zustands $\Pi = \{\pi, v_1, ..., v_n\}$ mit den Werten des Folgezustands $\Pi' = \{\pi', v_1', ..., v_{n'}\}$ in Beziehung. Die einfach gestrichenen Variablen Π' werden dazu verwendet, die *"neuen"* Werte der Variablen von den *"alten"* zu unterscheiden. Die Transitionsrelation ρ_τ hat folgende Form:

$$\rho_\tau(\Pi, \Pi') : C_\tau(\Pi) \Rightarrow (v_1' = e_1) \wedge ... \wedge (v_n' = e_n) \wedge (\pi' = s_j).$$

$C_\tau(\Pi)$ gibt die Bedingung an, unter der die Transitionsrelation aktiv wird. e_1 bis e_n repräsentieren beliebige Ausdrücke über Π und s_j gibt den Folgezustand an. Die Transitionsrelation des Systems ρ_Φ ergibt sich durch das Produkt aller Transitionsrelationen ρ_τ:

$$\rho_\Phi = \bigwedge_{i=1,...,k} \rho_{\tau_i}$$

Ein Transitionssystem läßt sich als *erweiterter endlicher Automat* darstellen. Ein erweiterter endlicher Automat ist ein Zustandsdiagramm, bei dem boolesche Ausdrücke als Bedingungen für einen Übergang und auszuführende Datenoperationen an den Kanten notiert sind und bei dem jedem Schritt ein symbolischer Ablaufzustand zugeordnet wird. Zu beachten ist, daß für diese Klasse der Automaten nicht die Möglichkeit besteht, einen eindeutigen, minimalen Automaten zu konstruieren. Ein Beispiel für einen erweiterten endlichen Automaten gibt Abbildung 3.10.

Beispiel 3.8 *Für den erweiterten Zustandsautomaten aus Abbildung 3.10 ergibt sich ein Transitionssystem mit* $\Phi = \{\pi, a, b, c\}$, $\Sigma = \{S^a, S^b\}$ *und* S^a *als Startzustand. Die Zustandsübergänge werden durch die vier folgenden Transitionsrelationen beschrieben:*

$$\rho_1 : (\pi = S^a) \wedge (a > b) \Rightarrow (a' = a - b) \wedge (b' = b) \wedge (c' = c) \wedge (\pi' = S^b),$$

$$\rho_2 : (\pi = S^a) \wedge \neg(a > b) \Rightarrow (a' = a) \wedge (b' = b) \wedge (c' = a) \wedge (\pi' = S^b),$$

$$\rho_3 : (\pi = S^b) \wedge (a < b) \Rightarrow (a' = a) \wedge (b' = b - a) \wedge (c' = c) \wedge (\pi' = S^a),$$

$$\rho_4 : (\pi = S^b) \wedge \neg(a < b) \Rightarrow (a' = a) \wedge (b' = b) \wedge (c' = b) \wedge (\pi' = S^a)$$

Das Transitionssystem berechnet den größten gemeinsamen Teiler zweier Zahlen a und b.

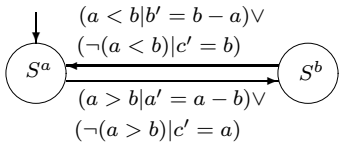


Abb. 3.10: Ein erweiterter endlicher Automat zur Bestimmung des GGT

Die Semantik einer normalisierten LLS-Beschreibung $\mathcal{B} = (\mathcal{S}_\mathcal{B}, S^{init}, \mathcal{V}_\mathcal{B})$ ist durch das synchrone Transitionssystem (Π, Σ, Φ, s_1) gegeben. Jede LLS-Beschreibung kann normalisiert werden. Da jedes Segment einer Beschreibung in Normalform einen Zeitschritt zur Ausführung benötigt, können die Segmente mit den Zuständen eines Transitionssystems bzw. eines erweiterten endlichen Automaten assoziiert werden. Für jedes Segment lassen sich eine oder, falls Verzweigungen auftreten, mehrere Transitionsrelationen aufstellen. Die Bedingung für das Aktivieren einer Transitionsrelation ergibt sich durch die Bedingungen der Verzweigungen. Der nachfolgende Zustand wird durch die Ausgangsmarken des Segments festgelegt. Es gilt $\Pi = \{\pi\} \cup \mathcal{V}_\mathcal{B}$, $\Sigma = \mathcal{S}_\mathcal{B}$ und $s_1 = S^{init}$.

Beispiel 3.9 *Wird die Beschreibung $\mathcal{B}$ aus Abbildung 3.5(a) normalisiert, ergibt sich $\mathcal{B}'$ aus 3.5(b). $\mathcal{B}'$ entspricht dem Transitionssystem, das in Beispiel 3.8 dargestellt wird. Der entsprechende Zustandsautomat wird in Abbildung 3.10 dargestellt.*

3.5 Übersetzung nach VHDL

Im Gegensatz zu VHDL existiert zur Zeit für LLS kein Simulator. Im Rahmen dieser Arbeit wurde aber ein Übersetzer entwickelt, der LLS-Beschreibungen nach VHDL übersetzt. So können einerseits die existierenden Simulationswerkzeuge, andererseits aber auch kommerzielle Synthese- und Verifikationswerkzeuge in den Entwurfsfluß einbezogen werden. Schwierigkeiten ergeben sich durch:

- die Modellierung des Zeitverhaltens von LLS und

- die Modellierung von Speichern.

3.5.1 Modellierung des Zeitverhaltens von LLS in VHDL

LLS ist eine axiomatisierte Hardwarebeschreibungssprache. Das Zeitverhalten von Ausdrücken in LLS ist durch die Axiomatisierung eindeutig festgelegt. Die Werte der Quellvariablen von Anweisungen werden durch den vorangehenden Zeitschritt festgelegt. So tauschen z.B. die Zuweisungen `(a←b,b←a)` die Werte der beiden Variablen aus. Aus diesem Grund werden alle Variablen in der LLS-Beschreibung als Signale in der VHDL-Beschreibung definiert. In der LLS-Beschreibung ist der Takt durch die Semantik der Sprache gegeben, so daß es in der VHDL-Beschreibung erforderlich wird, den Takt als ein zusätzliches Eingangssignal `clk` zu definieren. Über Anlegen eines geeigneten Taktsignals an `clk` kann das Zeitverhalten von LLS modelliert werden.

a)
```
L0 :
  IF p
    THEN x←y;
         L1;
    ELSE y←x;
         L2;
L1 :
  ...
```

b)
```
process
begin
  wait until clk'event and clk='1';
  if (reset='1') then
    control_state<=l0;
    p<='0';
    x<="00000000";
    y<="00000000";
  elsif control_state=l0 then
    if (p='1') then
      x<=y;
      control_state<=l1;
    else
      y<=x;
      control_state<=l2;
    end if;
  elsif ...
    ...
  end if;
end process;
```

Abb. 3.11: Übersetzung einer LLS-Beschreibung nach VHDL

Der Ablauf einer Berechnung einer LLS-Beschreibung ist durch die Anfangsmarke und die sich ergebenden Ausgangsmarken der Segmente festgelegt. In der VHDL-Beschreibung wird ein weiteres Signal `control_state` eingeführt, mit dessen Hilfe der Kontrollfluß modelliert wird. `control_state` speichert, welches Segment zu bearbeiten ist. An die Stelle der Ausgangsmarken treten Zuweisungen nach `control_state`. Um einen späteren Verifikationsschritt zu erleichtern, werden die Marken binär kodiert. Die Lesbarkeit der VHDL-Beschreibung läßt sich durch die Einführung von Konstanten erhöhen. In dem Beispiel von Abbildung 3.11 vertreten die Konstanten `l0`, `l1` bzw. `l2` die Bitvektoren "`00`", "`01`" bzw. "`10`". In der VHDL-Beschreibung wird ein Prozeß definiert, der eine `wait`-Anweisung und eine verschachtelte Auswahl enthält. Bei steigender Flanke des Taktes wird

die Auswahl ausgewertet. Die Auswahl verzweigt entsprechend `control_state`. Vor Berücksichtigung des `control_state`-Signals wird `reset` abgefragt, um, falls dieses Eingangssignal gesetzt ist, alle Signale auf die Initialisierung zurückzusetzen. Falls `reset` nicht bereits in der LLS-Beschreibung enthalten ist, wird `reset` automatisch eingeführt.

3.5.2 Modellierung von Speichern in LLS und in VHDL

In LLS werden Speicher als Arrays modelliert. Da dieser Datentyp für die Synthese ungeeignet ist, werden in der VHDL-Beschreibung Speicher strukturell beschrieben, indem die Daten-, Adreß- und Steuerleitungen des Speicherbausteins als Ein- und Ausgänge der Beschreibung definiert werden. Die Speicherbausteine werden nicht synthetisiert, statt dessen werden nach der Synthese bereits synthetisierte Komponenten verwendet.
Um zu vermeiden, daß Beschreibungen nicht synthetisiert werden können, bedarf es gewisser Einschränkungen. So können z.B. aus einem Speicher maximal zwei unterschiedliche Datenworte parallel ausgelesen und ein Datenwort abgespeichert werden. Verletzt eine LLS-Beschreibung die Einschränkungen nicht, werden die Speicherbausteine aus der Verhaltensbeschreibung extrahiert, um die modifizierte Beschreibung nach VHDL zu übersetzen. Der *Synopsys Design Compiler* [Syn98a] leitet aus der VHDL-Beschreibung eine strukturelle Repräsentation ab.
Multiplexer werden bei der Übersetzung nicht eingefügt. Erst der *Synopsys Design Compiler* generiert diese, falls verschiedene Register als Quelle oder zur Adressierung eines Speichers verwendet werden. Die Übersetzung einer LLS-Beschreibung setzt voraus, daß die jeweilige Adresse, an der in einem Speicher gelesen oder geschrieben werden soll, bereits durch den vorangehenden Takt bestimmt wird. Während einer Vorverarbeitung werden die zusätzlichen Zuweisungen eingefügt. Erschwerend kommt hinzu, daß LLS-Beschreibungen zyklisch sind und es sich um bedingte Speicherzugriffe handeln kann.

Beispiel 3.10 *Abbildung 3.12 zeigt ein Beispiel für die Übersetzung eines Speicherzugriffes. Die Registerbank* `rf` *wird in Abhängigkeit der Bedingung* `p` *an der Adresse* `a` *oder* `c` *beschrieben. Die Adresse* `a` *wird einen Schritt zuvor berechnet. Als Quelle des Schreibzugriffes dient unter der Bedingung, das* `p` *erfüllt ist,* `x`*, andernfalls wird der Wert von* `y` *abgespeichert.*
In der VHDL-Beschreibung werden drei weitere Ausgänge definiert: `rf_data`*,* `rf_adr` *und* `rf_wr_enable`*. Während über* `rf_data` *die abzuspeichernden Daten ausgegeben werden, ist* `rf_adr` *die anzusprechende Adresse zu entnehmen. Bei* `rf_wr_enable` *handelt es sich um ein Signal, das den Schreibzugriff steuert. In der VHDL-Beschreibung wird* `rf_adr` *einen Zeitschritt vor dem Schreibzugriff bestimmt. Es zeigt sich, daß sequentielle Datenabhängigkeiten beachtet und*

a)
```
L^x :
    a←a+b;
    L^y;
L^y :
    IF p
      THEN rf[a]←x;
      ELSE rf[c]←y;
    ...
```

b)
```
if (control_state=lx)
   then a⇐a+b;
        if (p='1')
          then rf_adr⇐a+b;
          else rf_adr⇐c;
        end if;
        control_state⇐ly;
elsif (control_state=ly)
   then rf_wr_enable⇐'0',
        if (p='1')
          then rf_daten⇐x;
          else rf_daten⇐y;
        end if;
...
```

Abb. 3.12: Übersetzung eines Speicherzugriffes

durch Forwarding (siehe Abschnitt 4.7) gelöst werden müssen. An die Stelle der Zuweisungen `rf[a]←x` *bzw.* `rf[c]←y` *treten die Anweisung* `rf_data⇐x` *bzw.* `rf_data⇐y`. *Eine weitere Zuweisung* `rf_wr_enable⇐'0'`, *die den Schreibzugriff ermöglicht, wird hinzugefügt.*

3.6 Zusammenfassung

Als textuelle Repräsentation wird die experimentelle Hardwarebeschreibungssprache LLS benutzt. Sie dient dazu, Systeme auf der algorithmischen und der Register-Transfer-Ebene darzustellen. Sie besitzt eine formale Semantik und eignet sich sowohl für die Synthese als auch für die Verifikation. Die Beschreibungen können normalisiert werden und repräsentieren erweiterte endliche Automaten. Im Gegensatz zu anderen Hardwarebeschreibungssprachen, wie z.B. VHDL, existiert für LLS kein Simulator. Um einerseits Beschreibungen simulieren zu können und andererseits kommerzielle Synthese- und Verifikationswerkzeuge in den Entwurfsfluß zu integrieren, wurde im Rahmen dieser Arbeit ein Übersetzer entwickelt, der LLS-Beschreibungen nach VHDL übersetzt.

Kapitel 4

Korrektheitserhaltende Transformationen

4.1 Einleitung

Die Anwendung korrektheitserhaltender Transformationen hat im Bereich des formal korrekten Hardwareentwurfs [Eve90] eine lange Tradition. Unter formaler Synthese werden Methoden verstanden, die das Syntheseergebnis innerhalb eines logischen Kalküls formal korrekt ableiten (siehe Abschnitt 2.2.2). Die Synthese ist auf die Anwendung formal definierter, korrektheitserhaltender Transformationen begrenzt, so daß nicht nur eine Implementierung für eine vorgegebene Spezifikation gewonnen wird, sondern gleichfalls ein Beweis der Korrektheit des Ergebnisses gegeben werden kann.
Der Kern des vorgestellten Synthesewerkzeugs besteht aus einer Menge von Transformationsregeln. Dabei wird zwischen hardware-spezifischen und allgemeingültigen bzw. logischen Transformationen unterschieden (siehe Abschnitt 2.2.2). Im folgenden wird eine Menge allgemeingültiger korrektheitserhaltender Transformationen definiert, die es erlauben, LLS-Beschreibungen formal korrekt zu verändern. Diese Transformationen lassen sich in zwei Klassen unterteilen:

- *formale Transformationen*, die der *Pfadäquivalenz* genügen und
- *kontext-abhängige Transformationen*, die im Sinne der *Berechnungsäquivalenz* korrektheitserhaltend sind.

Es werden im folgenden nur geschlossene Systeme bzw. Beschreibungen ohne Eingabe betrachtet. Besitzt eine Beschreibung eine Eingabe, wird es notwendig, über die Eingabebelegung in Abhängigkeit von der Zeit zu argumentieren. Die Eingabe kann als eine Funktion interpretiert werden, die die Belegung auf die Zeit abbildet. Zur Vereinfachung der Darstellung wird ohne Beschränkung der

Allgemeinheit auf Eingaben verzichtet.
Bevor die Abschnitte 4.6 bzw. 4.7 die Transformationen im einzelnen darstellen, werden die Begriffe der *Pfadäquivalenz*, der *Berechnungsäquivalenz* und der *Transformationsäquivalenz* definiert. Betrachtungen der Vollständigkeit und eine Zusammenfassung beenden das Kapitel.

4.2 Vorbemerkung

LLS-Beschreibungen $\mathcal{B} = (\mathcal{S}_\mathcal{B}, S^{init}, \mathcal{V}_\mathcal{B})$ (siehe Definition 3.1 Abschnitt 3.2) können durch Anwenden von Transformationen formal korrekt verändert werden. Die Menge aller Transformationen wird im folgenden mit Φ, die der formalen mit Φ_f und die der kontext-abhängigen mit Φ_k bezeichnet. $\mathcal{B} \xrightarrow{\phi} \mathcal{B}'$ gibt an, daß die Beschreibung $\mathcal{B}'$ durch Anwenden der Transformation $\phi \in \Phi$ aus $\mathcal{B}$ hervorgegangen ist.

Beispiel 4.1 *Abbildung 4.1(a) gibt ein Beispiel für eine Beschreibung* $\mathcal{B} = (\{S^a, S^b\}, S^a, \{a, b, c\})$ *mit den Segmenten* S^a *und* S^b *und den Variablen a, b und c, die den größten gemeinsamen Teiler zweier Zahlen berechnet.*

a) $\mathcal{B} = (\{S^a, S^b\}, S^a, \{a, b, c\})$ mit den Segmenten

```
Sa :
  IF a>b
    THEN a←a-b;
    ELSE c←a;
  Sb;
Sb :
  IF a<b
    THEN b←b-a;
    ELSE c←b;
  Sa;
```

b) $\mathcal{B}' = (\{S^a, S^b\}, S^a, \{a, b, c\})$ mit den Segmenten

```
Sa :
  IF ¬(a>b)
    THEN c←a;
    ELSE a←a-b;
  Sb;
Sb :
  IF a<b
    THEN b←b-a;
    ELSE c←b;
  Sa;
```

Abb. 4.1: Beispiel einer formalen Transformation

Das Anwenden einer formalen Transformation $\phi_f \in \Phi_f$ *auf die Beschreibung* $\mathcal{B}$ *in Abbildung 4.1(a) erzeugt* $\mathcal{B}'$ *in 4.1(b):* $\mathcal{B} \xrightarrow{\phi_f} \mathcal{B}'$. *In Segment* S^a *wird die Bedingung der if-then-else-Anweisung negiert und die Anweisungen des then- und else-Zweiges vertauscht. Die Transformation ist korrektheitserhaltend, so daß* $\mathcal{B}'$ *wie* $\mathcal{B}$ *den größten gemeinsamen Teiler zweier Zahlen berechnet.*

Eine *Konfiguration* $K^\mathcal{B}_{S^x,i} : \mathcal{B}\langle S^x, i; w_1, ..., w_n\rangle$ einer Beschreibung $\mathcal{B}$ gibt an, daß in der Beschreibung $\mathcal{B}$ das Segment $S^x \in \mathcal{S}_\mathcal{B}$ aktiv ist und die Variable $v_j \in \mathcal{V}_\mathcal{B}$ die Werte $w_j \in \mathcal{W}_{v_j}$ mit $1 \leq j \leq n$ nach Ausführen von i Zeitschritten annimmt. $K^\mathcal{B}_{S^x,0} : \mathcal{B}\langle S^x, 0; w_1, ..., w_n\rangle$ beschreibt die Initialisierung des Segments $S^x \in \mathcal{S}_\mathcal{B}$. Wird das Ende eines Berechnungspfades erreicht, ergibt sich $K^\mathcal{B}_{S^x,Ende} : \mathcal{B}\langle S^x, Ende; w_1, ..., w_n\rangle$ als Konfiguration des Segments $S^x \in \mathcal{S}_\mathcal{B}$. Die

Menge der betrachteten Variablen kann durch $K^{\mathcal{B}_V}_{S^x,i} : \mathcal{B}_V\langle S^x, i; w_1, ..., w_n\rangle$ auf eine Untermenge $V \subseteq \mathcal{V}_\mathcal{B}$ beschränkt werden.

Beispiel 4.2 *Berechnet die Beschreibung $\mathcal{B}$ für die Zahlen* 6 *und* 4 *den größten gemeinsamen Teiler, startet die Berechnung mit der folgenden Konfiguration: $K^{\mathcal{B}}_{S^a,0} : \mathcal{B}_{\{a,b,c\}}\langle S^a, 0; 6, 4, -\rangle$. Da der Wert der Variablen c erst im Laufe der Berechnung ermittelt wird, ist c zu Beginn unbestimmt. Zwei Zeitschritte später ergibt sich daraus die Konfiguration: $K^{\mathcal{B}}_{S^b,1} : \mathcal{B}_{\{a,b,c\}}\langle S^b, 1; 2, 2, -\rangle$.*

Eine LLS-Beschreibung führt eine *Berechnung* aus. Eine *Berechnung* $K^{\mathcal{B}_V}_{S^x,i} \overset{k}{\vdash} K^{\mathcal{B}_V}_{S^{x'},j}$ bzw. $\mathcal{B}_V\langle S^x, i; w_1, ..., w_n\rangle \overset{k}{\vdash} \mathcal{B}_V\langle S^{x'}, j; w'_1, ..., w'_n\rangle$ einer Beschreibung $\mathcal{B}$ stellt einen Konfigurationswechsel dar, der in k Zeitschritten ausgehend von der Konfiguration $K^{\mathcal{B}_V}_{S^x,i}$ die Beschreibung in die Konfiguration $K^{\mathcal{B}_V}_{S^{x'},j}$ überführt.

Beispiel 4.3 *Die Beschreibung $\mathcal{B}$ berechnet den größten gemeinsamen Teiler der Zahlen* 6 *und* 4 *folgendermaßen:*

$$\mathcal{B}_{\{a,b,c\}}\langle S^a, 0; 6, 4, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^a, 1; 2, 4, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^b, 1; 2, 2, -\rangle \overset{1}{\vdash}$$
$$\mathcal{B}_{\{a,b,c\}}\langle S^a, 1; 2, 2, 2\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^b, 1; 2, 2, 2\rangle.$$

Da zu Beginn `a>b` *gilt, wird im ersten Berechnungsschritt die Zuweisung* `a←a-b` *ausgeführt, so daß die Variable* `a` *den Wert* 2 *annimmt. S^b führt im nächsten Zeitschritt* `b←b-a` *aus, da* `a<b` *gilt. Für* `b` *ergibt sich ebenfalls der Wert* 2*. Nach erneutem Wechsel des Segments wird der Wert von* `a` *der Variablen* `c` *durch* `c←a` *zugewiesen. Die Berechnung terminiert nicht, weil es sich um ein reaktives System handelt. Die Werte der Variablen ändern sich aber nicht mehr.*

4.3 Pfadäquivalenz

Formale Transformationen (siehe Abschnitt 4.6) verändern den Kontrollfluß einer Beschreibung.

Beispiel 4.4 *In Abbildung 4.1 wird die Bedingung einer if-then-else-Anweisung negiert, was ein Vertauschen der Ausführungszweige bedingt. Zeitliche Eigenschaften der Beschreibung ändern sich nicht, so daß die Berechnungen von $\mathcal{B}$ und $\mathcal{B}'$ identisch sind. Der Kontrollfluß hat sich jedoch verändert, da in $\mathcal{B}$ die Bedingung* `a>b` *und in $\mathcal{B}'$* `¬(a>b)` *auszuwerten sind. Die Beschreibungen sind daher pfadäquivalent und die angewendete Transformation ist im Sinne der Pfadäquivalenz korrektheitserhaltend.*

Formale Transformationen erhalten die Pfadäquivalenz. Pfadäquivalenz meint, daß bei paralleler Ausführung zweier Beschreibungen die von den gemeinsamen Variablen gespeicherten Werte in jedem Schritt gleich sind bzw. daß sie identische Berechnungen ausführen. Formal ist die Pfadäquivalenz zweier Beschreibungen wie folgt definiert:

Definition 4.1 (Pfadäquivalenz)
Zwei Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ sind pfadäquivalent ($\mathcal{B} \cong|_V \mathcal{B}'$), wenn die bei der Ausführung der beiden Beschreibungen erzeugten Folgen von Werten einer Untermenge gemeinsamer Variablen $V \subseteq \mathcal{V}_\mathcal{B} \cap \mathcal{V}'_\mathcal{B}$ in jedem Schritt übereinstimmen:

$$\mathcal{B} \cong|_V \mathcal{B}' \Longleftrightarrow \left\{ \begin{array}{l} \forall k \in \mathcal{N} : \forall i \in [1:n] : \forall w_i \in \mathcal{W}_{v_i} \text{ mit } v_i \in V \subseteq \mathcal{V}_\mathcal{B} \cap \\ \mathcal{V}'_\mathcal{B} \ : \ \mathcal{B}_V\langle S^{init}, 0; w_1, ..., w_n\rangle \overset{k}{\vdash} \mathcal{B}_V\langle S^x, j; w'_1, ..., w'_n\rangle \Rightarrow \\ \mathcal{B}'_V\langle S^{init}, 0; w_1, ..., w_n\rangle \overset{k}{\vdash} \mathcal{B}'_V\langle S^{x'}, j'; w'_1, ..., w'_n\rangle \end{array} \right\}$$

Beispiel 4.5 *In Abbildung 4.2 berechnen alle drei Beschreibungen den größten gemeinsamen Teiler zweier Zahlen. Initialisiert mit den Zahlen 6 und 4 ergeben sich für Beschreibung $\mathcal{B}$ folgende Berechnungen:*

$$\mathcal{B}_{\{a,b,c\}}\langle S^a, 0; 6, 4, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^a, 1; 6, 4, 0\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^a, 2; 2, 4, 0\rangle \overset{1}{\vdash}$$
$$\mathcal{B}_{\{a,b,c\}}\langle S^a, 1; 2, 4, 0\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^a, 2; 2, 2, 0\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,c\}}\langle S^a, 1; 2, 2, 2\rangle$$

und für $\mathcal{B}'$:

$$\mathcal{B}'_{\{a,b,c\}}\langle S^a, 0; 6, 4, -\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,c\}}\langle S^a, 1; 6, 4, 0\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,c\}}\langle S^b, 1; 2, 4, 0\rangle \overset{1}{\vdash}$$
$$\mathcal{B}'_{\{a,b,c\}}\langle S^a, 1; 2, 4, 0\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,c\}}\langle S^b, 1; 2, 2, 0\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,c\}}\langle S^a, 1; 2, 2, 2\rangle$$

Es läßt sich zeigen, daß $\mathcal{B} \cong|_{\{a,b,c\}} \mathcal{B}'$ erfüllt ist. Für die Beschreibung $\mathcal{B}''$ ergibt sich folgende Berechnung:

$$\mathcal{B}''_{\{a,b,c\}}\langle S^a, 0; 6, 4, -\rangle \overset{1}{\vdash} \mathcal{B}''_{\{a,b,c\}}\langle S^a, 1; 2, 4, -\rangle \overset{1}{\vdash} \mathcal{B}''_{\{a,b,c\}}\langle S^a, 2; 2, 4, 0\rangle \overset{1}{\vdash}$$
$$\mathcal{B}''_{\{a,b,c\}}\langle S^a, 1; 2, 2, 0\rangle \overset{1}{\vdash} \mathcal{B}''_{\{a,b,c\}}\langle S^a, 2; 2, 2, 0\rangle \overset{1}{\vdash} \mathcal{B}''_{\{a,b,c\}}\langle S^a, 1; 2, 2, 2\rangle$$

Die Beschreibungen $\mathcal{B}$ und $\mathcal{B}''$ bzw. $\mathcal{B}'$ und $\mathcal{B}''$ sind somit nicht pfadäquivalent, obwohl auch $\mathcal{B}''$ den größten gemeinsamen Teiler korrekt bestimmt.

a) $\mathcal{B} = (\{S^a, S^b\}, S^a, \{a, b, c\})$ mit dem Segment

```
Sa :
  IF ¬(a=b)
    THEN IF a>b
           THEN c←0;
                a←a-b;
                Sa;
         ELSIF ¬a>b
           THEN c←0;
                b←b-a;
                Sa;
    ELSE c←a;
         SEnde;
```

b) $\mathcal{B}' = (\{S^a, S^b, S^c\}, S^a, \{a, b, c\})$ mit den Segmenten

```
Sa :
  IF a=b
    THEN c←a;
         SEnde;
    ELSE (c←0,
         p←a>b);
         Sb;
Sb :
  IF ¬p
    THEN b←b-a;
         Sa;
    ELSE a←a-b;
         Sa;
```

c) $\mathcal{B}'' = (\{S^a, S^b\}, S^a, \{a, b, c\})$ mit dem Segment

```
Sa :
  IF ¬(a=b)
    THEN IF a>b
           THEN a←a-b;
         ELSIF a<b
           THEN b←b-a;
         c←0;
         Sa;
    ELSE c←a;
         SEnde;
```

Abb. 4.2: GGT als Beispiel für die Pfadäquivalenz

Formale Transformationen erhalten die Pfadäquivalenz. Somit läßt sich der folgende Satz formulieren:

Satz 4.1 *Formale Transformationen Φ_f sind korrektheitserhaltend im Sinne der Pfadäquivalenz. Wird eine Beschreibung durch die Anwendung einer formalen*

Transformation verändert, so sind die ursprüngliche und die transformierte Beschreibung pfadäquivalent: $\forall \phi \in \Phi_f : \forall \mathcal{B} : \mathcal{B} \xrightarrow{\phi} \mathcal{B}' \Rightarrow \mathcal{B} \cong \mathcal{B}'$

4.4 Berechnungsäquivalenz

Im Gegensatz zu formalen Transformationen haben kontext-abhängige Transformationen (siehe Abschnitt 4.7) keinen Einfluß auf den Kontrollfluß, sondern parallelisieren bzw. serialisieren Folgen sequentieller Anweisungen innerhalb eines Segments.

Beispiel 4.6 *Die Beschreibungen* $\mathcal{B}$ *und* $\mathcal{B}''$ *in Abbildung 4.2 unterscheiden sich dadurch, daß die Zuweisung* `c←0` *vor bzw. nach der bedingten Ausführung von* `a←a-b` *und* `b←b-a` *in dem Segment* S^a *realisiert wird. Dennoch werden die gleichen Werte für die gemeinsamen Variablen berechnet. Somit sind* $\mathcal{B}$ *und* $\mathcal{B}''$ *berechnungsäquivalent. Durch Anwendung kontext-abhängiger Transformationen, die die Berechnungsäquivalenz erhalten, ist es möglich,* $\mathcal{B}$ *in* $\mathcal{B}''$ *bzw.* $\mathcal{B}''$ *in* $\mathcal{B}$ *umzuwandeln.*

Während die Pfadäquivalenz die Äquivalenz von Beschreibungen charakterisiert, ist die Berechnungsäquivalenz über Segmenten definiert. Die Berechnungsäquivalenz von Segmenten läßt keine Aussage zu, ob die nachfolgenden Segmente ebenfalls berechnungsäquivalent sind, sondern sagt aus, daß die von zwei Segmenten ausgeführten Berechnungen, beginnend mit der Übernahme des Kontrollflusses und endend an einer Ausgangsmarke, gleiche Endergebnisse liefern.

Definition 4.2 (Berechnungsäquivalenz)
Zwei Segmente $S^x \in \mathcal{S}_\mathcal{B}$ *und* $S^{x'} \in \mathcal{S}_{\mathcal{B}'}$ *sind berechnungsäquivalent* ($S^x \simeq|_V S^{x'}$), *falls bei identischer Initialisierung ihre Ausführung gleiche Endergebnisse einer Untermenge gemeinsamer Variablen* $V \subseteq \mathcal{V}_\mathcal{B} \cap \mathcal{V}'_\mathcal{B}$ *ergibt:*

$$S^x \simeq|_V S^{x'} \Longleftrightarrow \left\{ \begin{array}{l} \forall i \in [1:n] : \forall w_i \in \mathcal{W}_{v_i} \text{ mit } v_i \in V \subseteq \\ \mathcal{V}_\mathcal{B} \cap \mathcal{V}'_\mathcal{B} : \exists k, l \in \mathcal{N} : \mathcal{B}_V\langle S^x, 0; w_1, ..., w_n\rangle \stackrel{k}{\vdash} \\ \mathcal{B}_V\langle S^x, Ende; w'_1, ..., w'_n\rangle \Rightarrow \mathcal{B}'_V\langle S^{x'}, 0; w_1, ..., w_n\rangle \stackrel{l}{\vdash} \\ \mathcal{B}'_V\langle S^{x'}, Ende; w'_1, ..., w'_n\rangle \end{array} \right\}$$

Beispiel 4.7 *In Abbildung 4.3 berechnen sowohl das Segment* $S^a \in \mathcal{S}_\mathcal{B}$ *als auch* $S^x \in \mathcal{S}_{\mathcal{B}'}$ *die Formel* $(a+b)^3$. *Für die Zahlen* 6 *und* 4 *ergeben sich die folgenden Berechnungen:*

$\mathcal{B}_{\{a,b,d\}}\langle S^a, 0; 6, 4, -\rangle \stackrel{1}{\vdash} \mathcal{B}_{\{a,b,d\}}\langle S^a, 1; 6, 4, -\rangle \stackrel{1}{\vdash} \mathcal{B}_{\{a,b,d\}}\langle S^a, 2; 6, 4, 100\rangle \vdash$
$\mathcal{B}_{\{a,b,d\}}\langle S^a_{S^b}, Ende; 6, 4, 1000\rangle$

und

$\mathcal{B}'_{\{a,b,d\}}\langle S^x, 0; 6, 4, -\rangle \stackrel{1}{\vdash} \mathcal{B}'_{\{a,b,d\}}\langle S^x, 1; 6, 4, 100\rangle \stackrel{1}{\vdash} \mathcal{B}'_{\{a,b,d\}}\langle S^x, Ende; 6, 4, 1000\rangle$

Die Variable c *bleibt unbeachtet, da sie nicht zu den gemeinsamen Variablen der Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ gehört. Während S^a das Ergebnis in drei Zeitschritten berechnet, benötigt S^x hingegen nur zwei Zeitschritte. Ein weiterer Unterschied besteht darin, daß die Variable* d *in $\mathcal{B}'$ bereits im ersten Zeitschritt verwendet wird. Da die gemeinsamen Variablen nach Ausführung von S^a und S^x identische Werte speichern, sind S^a und S^x berechnungsäquivalent: $S^a \simeq|_{\{a,b,d\}} S^x$. $\mathcal{B}$ und $\mathcal{B}'$ sind jedoch nicht pfadäquivalent, weil S^a und S^x die Berechnungen in unterschiedlichen Schritten ausführen.*

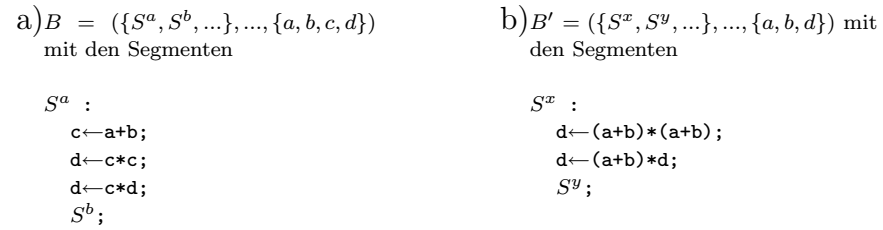


Abb. 4.3: Beispiel für die Berechnungsäquivalenz

Kontext-abhängige Transformationen genügen der Berechnungsäquivalenz, so daß sich der folgende Satz formulieren läßt:

Satz 4.2 *Kontext-abhängige Transformationen Φ_k sind korrektheitserhaltend im Sinne der Berechnungsäquivalenz. Wird eine Beschreibung durch Anwendung einer kontext-abhängigen Transformation verändert, so sind die ursprüngliche und die transformierte Beschreibung berechnungsäquivalent: $\forall \phi \in \Phi_k : \forall \mathcal{B} : \mathcal{B} \xrightarrow{\phi} \mathcal{B}' \Rightarrow \mathcal{B} \simeq \mathcal{B}'$*

4.5 Transformationsäquivalenz

Durch Anwendung einer Folge formaler und kontext-abhängiger Transformationen verändern sich sowohl der Kontrollfluß als auch die zeitlichen Eigenschaften einer Beschreibung. So sind die ursprüngliche und die transformierte Beschreibung weder pfadäquivalent, noch finden sich berechnungsäquivalente Segmente.

Beispiel 4.8 *In Abbildung 4.2 sind die Beschreibungen $\mathcal{B}'$ und $\mathcal{B}''$ nicht pfadäquivalent, da die Variable* c *in unterschiedlichen Zeitschritten der Berechnungen beschrieben wird. S^a in Beschreibung $\mathcal{B}''$ ist aber auch zu keinem Segment in $\mathcal{B}'$ berechnungsäquivalent. Einerseits wird in S^a in $\mathcal{B}'$ keine Subtraktion* a-b *bzw.* b-a *ausgeführt, andererseits ist die Verzweigung in S^b von der Variablen* p *abhängig, die in dem vorangehenden Segment bestimmt wird. Eine Berechnungsäquivalenz ist somit nicht gegeben, da sie nur für bestimmte, nicht aber für beliebige Werte von* p *gezeigt werden kann.*

Die Äquivalenz läßt sich über die Folge der angewendeten Transformationen zeigen. Die ursprüngliche und die transformierte Beschreibung sind somit im Sinne einer Transformationsäquivalenz korrektheitserhaltend.

Definition 4.3 (Transformationsäquivalenz)
Zwei Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ sind transformationsäquivalent ($\mathcal{B} \doteq \mathcal{B}'$), wenn eine Folge korrektheitserhaltender Transformationen $\phi_1, ..., \phi_n \in \Phi$ existiert, die $\mathcal{B}$ nach $\mathcal{B}'$ überführt:

$$\mathcal{B} \doteq \mathcal{B}' \Longleftrightarrow \exists \phi_1, ..., \phi_n \in \Phi : \mathcal{B} \xrightarrow{\phi_1} ... \xrightarrow{\phi_n} \mathcal{B}'$$

Beispiel 4.9 *Die Beschreibungen $\mathcal{B}'$ und $\mathcal{B}''$ in Abbildung 4.2 sind weder pfad- noch berechnungsäquivalent. Es existiert eine Folge formaler und kontext-abhängiger Transformationen, mit der $\mathcal{B}'$ in $\mathcal{B}''$ oder umgekehrt umgewandelt werden kann. Aus diesem Grund sind die beiden Beschreibungen transformationsäquivalent.*

4.6 Formale Transformationen

Formale Transformationen verändern den Kontrollfluß einer Beschreibung. Diese Klasse der Transformationen basiert auf einer Theorie von Mikroprogrammschemata nach Glushko [Glu65]. Eine Theorie von Programmschemata reflektiert Eigenschaften, die unabhängig von bestimmten Programmen sind.

Beispiel 4.10 *Durch Anwenden des Axioms A0 (siehe Abbildung 4.4) auf eine if-then-else-Anweisung einer Beschreibung werden die Bedingung der if-then-else-Anweisung negiert und die Anweisungen des then- und des else-Zweiges vertauscht, so daß aus* `IF p THEN a; ELSE b;` *die Anweisung* `IF ¬p THEN b; ELSE a;` *entsteht. Diese Transformation wird unabhängig davon ausgeführt, welche Anweisungen durch* `a`, `b` *und* `p` *repräsentiert werden.*

Die Axiome aus Abbildung 4.4 gehen auf [Glu65] zurück und geben an, wie if-then-else-Strukturen verändert werden können. Die Axiome gelten unabhängig davon, an welcher Stelle sie in einer Beschreibung angewendet werden und welche Strukturen in der if-then-else-Anweisung enthalten sind.
Axiom A7 ermöglicht es, Anweisungen in eine if-then-else-Klausel hineinzuziehen. `{a}p` wird *virtuelles Prädikat* genannt und gibt an, daß die Bedingung `p` nach der virtuellen Ausführung der Anweisung `a` auszuwerten ist. Verändert der Ausdruck `a` keine der in `p` verwendeten Variablen, kann `{a}` entfallen. Andernfalls müssen die gemeinsamen Variablen in `p` entsprechend substituiert werden.

Beispiel 4.11 *Wird A7 auf* `x←x+1;IF x=0 THEN ... ELSE ...` *angewendet, ergibt sich* `IF {x←x+1}x=0 THEN x←x+1;... ELSE x←x+1;...` *Das virtuelle*

Axiom (A0) : $\mathcal{B}\,(\texttt{a;}) \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE a;}\end{pmatrix}$

Axiom (A1) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE b;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF }\neg\texttt{p}\\ \texttt{THEN b;}\\ \texttt{ELSE a;}\end{pmatrix}$

Axiom (A2) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN IF q}\\ \qquad\texttt{THEN a;}\\ \qquad\texttt{ELSE b;}\\ \texttt{ELSE c;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF (p}\wedge\texttt{q)}\\ \texttt{THEN a;}\\ \texttt{ELSE IF (p}\wedge\neg\texttt{q)}\\ \qquad\texttt{THEN b;}\\ \qquad\texttt{ELSE c;}\end{pmatrix}$

Axiom (A3) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE IF q}\\ \qquad\texttt{THEN b;}\\ \qquad\texttt{ELSE c;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\vee\texttt{q}\\ \texttt{THEN IF p}\\ \qquad\texttt{THEN a;}\\ \qquad\texttt{ELSE b;}\\ \texttt{ELSE c;}\end{pmatrix}$

Axiom (A4) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\wedge\texttt{q}\\ \texttt{THEN a;}\\ \texttt{ELSE b;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN IF q}\\ \qquad\texttt{THEN a;}\\ \qquad\texttt{ELSE b;}\\ \texttt{ELSE b;}\end{pmatrix}$

Axiom (A5) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\vee\texttt{q}\\ \texttt{THEN a;}\\ \texttt{ELSE b;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE IF q}\\ \qquad\texttt{THEN a;}\\ \qquad\texttt{ELSE b;}\end{pmatrix}$

Axiom (A6) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE b;}\\ \texttt{c;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;c;}\\ \texttt{ELSE b;c;}\end{pmatrix}$

Axiom (A7) : $\mathcal{B}\begin{pmatrix}\texttt{a;}\\ \texttt{IF p}\\ \texttt{THEN b;}\\ \texttt{ELSE c;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF \{a\}p}\\ \texttt{THEN a;b;}\\ \texttt{ELSE a;c;}\end{pmatrix}$

Axiom (A8) : $\mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN a;}\\ \texttt{ELSE b;}\end{pmatrix} \quad \cong \quad \mathcal{B}\begin{pmatrix}\texttt{IF p}\\ \texttt{THEN IF p}\\ \qquad\texttt{THEN a;}\\ \qquad\texttt{ELSE c;}\\ \texttt{ELSE b;}\end{pmatrix}$

Abb. 4.4: Formale Transformationen

Prädikat {x←x+1}x=0 *kann durch* x+1=0 *ersetzt werden, so daß sich* IF x+1=0 THEN x←x+1;... ELSE x←x+1;... *ergibt.*

Virtuelle Prädikate lassen sich durch Anwendung kontext-abhängiger Transformationen, insbesondere *Forwarding*, eliminieren, so daß nach Entfernen des virtuellen Prädikats die ursprüngliche und die transformierte Beschreibung nicht mehr pfad- sondern berechnungsäquivalent sind.
Neue Segmente können durch Anwenden der Regel aus Abbildung 4.5 eingeführt werden, es ist aber auch möglich, zwei Segmente zu einem zusammenzufassen. Die Regel besagt, daß ein Segment S und seine Marke L äquivalent sind und wechselseitig ersetzt werden können.

$$\textbf{Regel (R1) :} \quad \frac{L \ : \ S}{\mathcal{B}(L) \ \cong \ \mathcal{B}(S)}$$

Abb. 4.5: Erzeugen und Verschmelzen von Segmenten

Beispiel 4.12 *Regel R1 erlaubt es, neue Segmente einzuführen. In Abbildung 4.6 wird durch Anwendung von R1 auf 4.6(a) das Segment* Lx *erzeugt, so daß sich 4.6(b) ergibt. Es können aber auch zwei Segmente zu einem zusammengefaßt werden. Wird R1 auf 4.6(b) angewendet, ergibt sich 4.6(a).* Lx *wird nur entfernt, falls das Segment nach Anwenden der Regel unerreichbar ist.* a *und* b *repräsentieren beliebige Anweisungen.*

a)
```
L0 : a; b; L1;
```

b)
```
L0 : a; Lx;
Lx : b; L1;
```

Abb. 4.6: Anwendung der Regel R1

Wie bereits das vorangehende Beispiel zeigt, definieren die Axiome A0 bis A8 und die Regel R1 die zugehörigen inversen Transformationen. Invers meint, daß zu einer Transformation eine Rücktransformation existiert. Wird eine Transformation ausgeführt, so kann die ursprüngliche Sequenz durch Anwenden der Rücktransformation wieder hergestellt werden.

4.7 Kontext-abhängige Transformationen

Kontext-abhängige Transformationen erlauben es, Anweisungen zu parallelisieren bzw. zu serialisieren. Das nachfolgende Beispiel zeigt, daß beim Verändern der zeitlichen Abfolge von Anweisungen Datenabhängigkeiten beachtet und Konflikte gelöst werden müssen.

Beispiel 4.13 *In Abbildung 4.7(a) wird die Zuweisung* `y←x*a` *im zweiten Zeitschritt, in 4.7(b) im ersten ausgeführt. Die Segmente L^a und $L^{a'}$ sind nicht berechnungsäquivalent aufgrund von Datenabhängigkeiten, da bei Initialisierung mit 2, 3, 0 und 6 $\mathcal{B}$ und $\mathcal{B}'$ verschiedene Ergebnisse berechnen:*

$$\mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, 0; 2, 3, 0, 6, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, 1; 2, 3, 5, 6, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, Ende; 2, 3, 5, 10, 11\rangle$$

und $\mathcal{B}'$:

$$\mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, 0; 2, 3, 0, 6, -\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, 1; 2, 3, 5, 0, -\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, Ende; 2, 3, 5, 0, 5\rangle$$

Wird die Zuweisung `y←x*a` *in 4.7(a) einen Schritt vorgezogen, ist zu beachten, daß die Variable* `y` *in* `z←x+y` *benutzt wird. Darüberhinaus wird die Variable* `x`*, die in* `y←x*a` *verwendet wird, einen Schritt zuvor berechnet. Die zeitliche Abfolge von Anweisungen kann nur verändert werden, wenn Datenabhängigkeiten zuvor gelöst worden sind.*

a) $\mathcal{B} = (\{L^a, L^b, ...\}, ..., \{a, b, x, y, z\})$ mit den Segmenten

L^a:

```
x←a+b;
(y←x*a,z←x+y);
L^b;
```

b) $\mathcal{B}' = (\{L^{a'}, L^{b'}, ...\}, ..., \{a, b, x, y, z\})$ mit den Segmenten

$L^{a'}$

```
(x←a+b,y←x*a);
z←x+y;
L^b';
```

Abb. 4.7: Datenabhängigkeiten verhindern das Parallelisieren

Die folgenden Techniken erlauben das Lösen von Datenabhängigkeiten:

- *Einführen von Pipeline-Registern*: Zwei parallele Anweisungen sollen sequentiell ausgeführt werden, wobei die vorzuziehende Anweisung eine Variable als Ziel besitzt, die in der anderen als Quelle dient. Wird der alte Wert der betreffenden Variablen durch eine neu eingeführte Variable zwischengespeichert und in der Quelle durch diese substituiert, können die Anweisungen nacheinander ausgeführt werden.

```
(a←<Ausdruck>, b← ... op a op ...);
```
wird zu
```
(a←<Ausdruck>, ap←a); b← ... op ap op ...;.
```

- *Forwarding*: Zwei sequentielle Anweisungen werden parallelisiert, wobei die vorangehende den Wert einer Quellvariablen der nachfolgenden Anweisung berechnet. Wird diese Quellvariable durch den Ausdruck, der ihr in der ersten Anweisung zugeordnet wird, in der zweiten Anweisung ersetzt, können die beiden Anweisungen parallel ausgeführt werden.

a←<Ausdruck>; b← ... op a op ...;
wird zu
(a←<Ausdruck>, b← ... op <Ausdruck> op ...);.

- *Eliminieren redundanter Zuweisungen*: Zwei sequentielle Anweisungen besitzen die gleichen Zielvariablen. Falls **keine** weiteren Abhängigkeiten zwischen diesen Anweisungen bestehen, genügt es, die erste entfallen zu lassen und nur die zweite Anweisung auszuführen.

  a←<1.Ausdruck>; a←<2.Ausdruck>;
  wird zu
  a←<2.Ausdruck>;

  Bestehen weitere Abhängigkeiten, müssen diese zuvor mit den bereits vorgestellten Techniken gelöst werden.

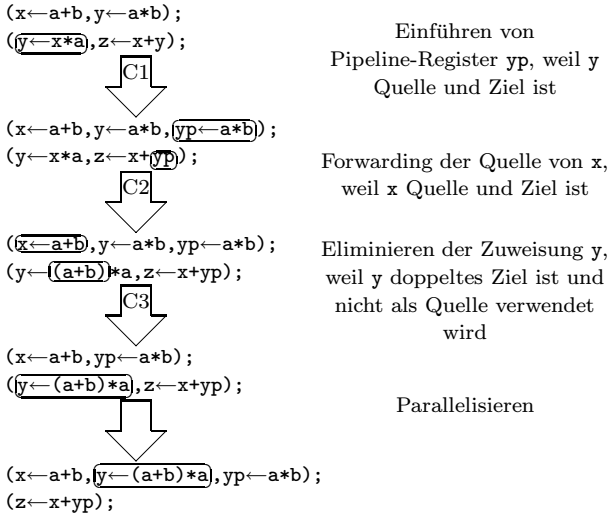


Abb. 4.8: Parallelisieren von Zuweisungen

Beispiel 4.14 *Abbildung 4.8 gibt ein Beispiel für die Anwendung der Techniken, mit deren Hilfe die Registerzuweisung* y←x*a *vorgezogen wird. Im Unterschied zu Abbildung 4.7(a) wird im ersten Zeitschritt die Zuweisung* y←a*b *zusätzlich realisiert.*
Da y *sowohl als Quelle in* z←x+y *als auch als Ziel in* y←x*a *verwendet wird, ist ein Pipeline-Register* yp *einzuführen. Im ersten Verarbeitungsschritt bekommt* yp *parallel zu* y *den Wert* a*b *zugewiesen. Im zweiten Schritt tritt* yp *in der Zuweisung* z←x+y *an die Stelle von* y. y←x*a *verwendet die Variable* x*, deren Wert einen Schritt zuvor berechnet wird. Die Zuweisungen lassen sich nur parallelisieren, wenn* x *durch die Quelle der Zuweisung* x←a+b *ersetzt wird. Die Zuweisung*

`y←a*b` *kann eliminiert werden, da der berechnete Wert keine Verwendung findet und* `y` *im anschließenden Schritt einen neuen Wert zugewiesen bekommt. Sie muß eliminiert werden, wenn* `y←(a+b)*a` *vorgezogen wird.*
Nach Durchführen der Transformationen sind die ursprüngliche und die transformierte Beschreibung berechnungsäquivalent. Im Vergleich zu Beispiel 4.13 ergeben sich die folgenden Berechnungen:

$$\mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, 0; 2, 3, 0, -, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, 1; 2, 3, 5, 6, -\rangle \overset{1}{\vdash} \mathcal{B}_{\{a,b,x,y,z\}}\langle L^a, \mathit{Ende}; 2, 3, 5, 10, 11\rangle$$

und

$$\mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, 0; 2, 3, 0, -, -\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, 1; 2, 3, 5, 10, -\rangle \overset{1}{\vdash} \mathcal{B}'_{\{a,b,x,y,z\}}\langle L^{a'}, \mathit{Ende}; 2, 3, 5, 10, 11\rangle$$

C1, C2 und C3 in Abbildung 4.8 stellen einen Bezug zu den im folgenden definierten Transformationen her.

Zur formalen Definition der kontext-abhängigen Transformationen werden verwendet:

Mengen:

- $\mathcal{A}$ als Menge der Anweisungen
- $\mathcal{V}$ als Menge der Variablen
- $\mathcal{EX}$ als Menge aller korrekten Ausdrücke über $\mathcal{V}$
- $\mathcal{D} \subseteq \mathcal{V}$ als Menge aller verwendeten Zielvariablen
- $\mathcal{S} \subseteq \mathcal{V}$ als Menge aller verwendeten Quellvariablen

Funktionen:

- $Z : \mathcal{A} \times \mathcal{D} \to \mathcal{A}$ bestimmt alle Zuweisungen aus $\mathcal{A}$ mit Zielvariablen in $\mathcal{D}$
- $D : \mathcal{A} \to \mathcal{V}$ berechnet alle Zielvariablen von Zuweisungen aus $\mathcal{A}$
- $S : \mathcal{A} \to \mathcal{V}$ ermittelt alle Variablen, die als Quelle in $\mathcal{A}$ dienen
- $Q : \mathcal{A} \times \mathcal{D} \to \mathcal{EX}$ liefert die Ausdrücke, die als Quelle der Anweisungen aus $\mathcal{A}$ mit den Zielen in $\mathcal{D}$ dienen

Schreibweisen:

- $\mathcal{M}//b \mapsto a \mathrel{\hat{=}}$ Ausdruck b ersetzt a in der Menge $\mathcal{M}$
- (a, b) bezeichnet die parallele und $a; b$ die sequentielle Komposition

Die Sequenz von Anweisungen $\mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3)$; kann zu $(\mathcal{A}_1, \mathcal{A}_2); \mathcal{A}_3$; transformiert werden, wenn keine Datenabhängigkeiten untereinander bestehen bzw. die Bernsteinschen Regeln [Ber66] erfüllt sind:

- $D(\mathcal{A}_1) \cap D(\mathcal{A}_2) = \emptyset$
- $D(\mathcal{A}_2) \cap S(\mathcal{A}_3) = \emptyset$
- $D(\mathcal{A}_1) \cap S(\mathcal{A}_2) = \emptyset$

Ist eine dieser drei Bedingungen nicht erfüllt, muß der Konflikt durch eine oder mehrere der folgenden Techniken gelöst werden:

Einführen von Pipeline-Registern :
Axiom (C1) : $\forall v \in D(\mathcal{A}_2) \cap S(\mathcal{A}_3) : \mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3); \simeq (\mathcal{A}_1, \mathcal{A}_2'); \mathcal{A}_3';$
mit $\mathcal{A}_2' = \mathcal{A}_2 \cup \{p_v \leftarrow v\}$ und $S(\mathcal{A}_3') = S(\mathcal{A}_3)//p_v \mapsto v$

Forwarding :
Axiom (C2) : $\forall v \in D(\mathcal{A}_1) \cap S(\mathcal{A}_2) : \mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3); \simeq (\mathcal{A}_1, \mathcal{A}_2'); \mathcal{A}_3;$
mit $S(\mathcal{A}_2') = S(\mathcal{A}_2)//Q(\mathcal{A}_1, \{v\}) \mapsto v$

Eliminieren redundanter Zuweisungen :
Axiom (C3) : $\forall v \in D(\mathcal{A}_1) \cap D(\mathcal{A}_2) : \mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3); \simeq (\mathcal{A}_1, \mathcal{A}_2'); \mathcal{A}_3;$
mit $\mathcal{A}_2' = \mathcal{A}_2 \backslash Z(\mathcal{A}_2, \{v\})$

Die Sequenz von Anweisungen $(\mathcal{A}_1, \mathcal{A}_2); \mathcal{A}_3;$ kann zu $\mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3);$ transformiert werden, wenn die folgenden Bedingungen erfüllt sind:

- $D(\mathcal{A}_2) \cap D(\mathcal{A}_3) = \emptyset,$
- $D(\mathcal{A}_1) \cap S(\mathcal{A}_2) = \emptyset.$
- $D(\mathcal{A}_2) \cap S(\mathcal{A}_3) = \emptyset,$

Wiederum löst das Entfernen überflüssiger Zuweisungen, Forwarding und das Einführen von Pipeline-Registern bestehende Konflikte:

Einführen von Pipeline-Registern :
Axiom (C4) : $\forall v \in D(\mathcal{A}_1) \cap S(\mathcal{A}_2) : (\mathcal{A}_1, \mathcal{A}_2); \mathcal{A}_3; \simeq \mathcal{A}_1'; (\mathcal{A}_2', \mathcal{A}_3);$
mit $\mathcal{A}_1' = \mathcal{A}_1 \cup \{p_v \leftarrow v\}$ und $S(\mathcal{A}_2') = S(\mathcal{A}_2)//p_v \mapsto v$

Forwarding :
Axiom (C5) : $\forall v \in D(\mathcal{A}_2) \cap S(\mathcal{A}_3) : (\mathcal{A}_1, \mathcal{A}_2); \mathcal{A}_3; \simeq \mathcal{A}_1; (\mathcal{A}_2, \mathcal{A}_3');$
mit $S(\mathcal{A}_3') = S(\mathcal{A}_3)//Q(\mathcal{A}_2, \{v\}) \mapsto v$

Entfernen überflüssiger Zuweisungen :
Axiom (C6) : $\forall v \in D(\mathcal{A}_2) \cap D(\mathcal{A}_3) : (\mathcal{A}_1, \mathcal{A}_2); \mathcal{A}_3; \simeq \mathcal{A}_1; (\mathcal{A}_2', \mathcal{A}_3);$
mit $\mathcal{A}_2' = \mathcal{A}_2 \backslash Z(\mathcal{A}_2, \{v\})$

Kontext-abhängige Transformationen erlauben es, azyklische Sequenzen von Anweisungen zu verändern. Neben Zuweisungen können auch if-then-else-Strukturen transformiert werden. Voraussetzung ist, daß der if-then-else-Ausdruck einen Zeitschritt zur Ausführung benötigt. Andernfalls müssen Anweisungen unter Anwendung formaler Transformationen verschoben werden, um die Transformation inner- oder außerhalb der if-then-else-Struktur durchzuführen.

Beispiel 4.15 *Die Anweisungen* `x←a+b` *und* `if x=0 then a←b-c else d←a` *werden zu* `(x←a+b,if (a+b)=0 then a←b-c else d←a)` *parallelisiert. Es ist ebenfalls möglich* `x←a+b` *in die if-then-else-Struktur hineinzuziehen (Axiom 7A), um im Anschluß die Zuweisungen zu parallelisieren:* `if (a+b)=0 then x←a+b; a←b-c else x←a+b;d←a`. *Das virtuelle Prädikat entfällt durch Substitution.*

Für kontext-abhängige Transformationen existieren keine inversen Transformationen. Zwar können mit Hilfe von C4 bis C6 Anweisungen, die mit C1 bis C3 vorgezogen worden sind, wieder verschoben werden. Die durch Forwarding ersetzten Ausdrücke, die eingeführten Pipeline-Register und die eliminierten Anweisungen lassen sich jedoch nicht mehr eindeutig bestimmen.

Beispiel 4.16 *Werden die zwei sequentiellen Zuweisungen* `a←b` *und* `d←a+c` *parallelisiert, ergibt sich durch Forwarding* `a←b,d←b+c`. *Das Parallelisieren von* `a←b` *und* `d←b+c` *führt zu dem gleichen Ergebnis, so daß eine* **eindeutige** *Invertierung der Transformation nicht möglich ist.*

4.8 Vollständigkeit der Transformationen

Im allgemeinen ist der Nachweis der Vollständigkeit eines Transformationssystems sehr schwierig. Die Vollständigkeit garantiert, daß der Entwurfsraum vollständig durch Anwenden der Transformationen abgesucht werden kann. Durch diese Eigenschaft wird ebenfalls ausgeschlossen, daß die Transformationen zu beschränkt sind. Ist das System nicht vollständig, können manuelle Eingriffe des Benutzers erforderlich werden.
Nach Vemuri [Vem90a] gelten zwei Beschreibungen als äquivalent, wenn sie die gleiche Spezifikation implementieren. Ausgehend von diesem Äquivalenzbegriff kann die Vollständigkeit eines Transformationssystems wie folgt definiert werden:

Definition 4.4 (Vollständigkeit)
Eine Menge von Transformationen ist vollständig, wenn für alle Paare äquivalenter Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ eine endliche Folge von Transformationen existiert, die $\mathcal{B}$ nach $\mathcal{B}'$ überführt.

Das in dieser Arbeit vorgestellte Transformationssystem ist vollständig, wenn für jedes Paar äquivalenter Beschreibungen die Transformationsäquivalenz gezeigt werden kann.
In [Vem90a] und [Vem90b] wird die Vollständigkeit einer Menge *struktureller Transformationen* der RT-Ebene bewiesen. Vemuri unterscheidet zwischen *strukturellen Transformationen*, die strukturelle Beschreibungen als Ergebnis haben, und *Verhaltenstransformationen*, die Verhaltensbeschreibungen transformieren.

Der Beweis der Vollständigkeit beschränkt sich auf strukturelle Transformationen. Unter der Voraussetzung, daß zu jeder Transformation eine inverse Transformation existiert und daß zu jeder Beschreibung eine eindeutige Normalform [Vem90b] angegeben werden kann, läßt sich die Vollständigkeit des Transformationssystems wie folgt beweisen:

> Für zwei äquivalente Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ existiert eine eindeutige Normalform $\mathcal{B}^{nf}$. Die Beschreibung $\mathcal{B}$ läßt sich durch die Transformationsfolge π und $\mathcal{B}'$ durch π' in die Normalform $\mathcal{B}^{nf}$ überführen: $\mathcal{B} \xrightarrow{\pi} \mathcal{B}^{nf}$ und $\mathcal{B}' \xrightarrow{\pi'} \mathcal{B}^{nf}$. Da zu jeder Transformation π eine inverse Transformation π^{-1} definiert ist, kann $\mathcal{B}$ durch $\mathcal{B} \xrightarrow{\pi} \mathcal{B}^{nf} \xrightarrow{\pi'^{-1}} \mathcal{B}'$ in $\mathcal{B}'$ umgewandelt werden. □

Für die in Abschnitt 4.6 und 4.7 vorgestellten Transformationen kann die Vollständigkeit entsprechend dem obigen Beweis nicht nachgewiesen werden, da diese das Verhalten einer Schaltung transformieren [Vem90a, Vem90b]. Einerseits sind die inversen Transformationen für die kontext-abhängige Transformationen nicht eindeutig, andererseits gibt es für eine LLS-Beschreibung keine eindeutige Normalform. Es kann nicht garantiert werden, daß zwei äquivalente LLS-Beschreibungen die gleiche Normalform besitzen. Die Vollständigkeit des Transformationssystems ist eine offene Frage. Die Anwendungsbeispiele legen dies jedoch nahe.

4.9 Formale Verifikation der Ergebnisse

Die formale Synthese basiert auf der Idee, daß die Korrektheit des Syntheseergebnisses gewährleistet werden kann, indem die Synthese auf das Anwenden korrektheitserhaltender Transformationen beschränkt wird. Die Folge der Transformationen dient als Beweis für die Korrektheit des Ergebnisses. Eine Überprüfung des Ergebnisses wird dennoch notwendig, da nicht ausgeschlossen werden kann, daß die Implementierung des verwendeten Synthesewerkzeugs fehlerhaft ist.
Viele formale Synthesewerkzeuge wie z.B. *LAMBDA/DIALOG* [FFFH90], *VERITAS* [HDL89] oder *HASH* [BS99] verwenden einen Theorembeweiser, um die Korrektheit der Transformationen sicherzustellen. Im Unterschied zu den genannten benutzt das *TUD Transformationswerkzeug (TUDT)* nur eine sehr kleine Menge allgemeingültiger Axiome und Regeln. Je kleiner der Transformationskern ist, desto größer ist die Sicherheit, daß keine Fehler bei der Implementierung auftreten. Einen weiteren Schutz gegen Implementierungsfehler stellt die geringe Komplexität der angewendeten Transformationen dar. Einen Beweis für die Korrektheit liefert aber erst die Post-Synthese-Verifikation.
Ein am Lehrstuhl entwickelter Äquivalenzprüfer [REH99, RHE99a, RHE99b]

weist die Korrektheit des Entwurfs nach. Er ist von dem *TUD Transformationswerkzeug (TUDT)* unabhängig und wird im Rahmen diese Arbeit verwendet, um die Berechnungsäquivalenz der ursprünglichen Beschreibung bzw. der Spezifikation und der durch den Transformationsprozeß abgeleiteten Beschreibung, auch Implementierung genannt, zu zeigen. Der Nachweis erfolgt über eine symbolische Simulation (siehe Abschnitt 2.2.3).
Um Fehler der Implementierung möglichst frühzeitig zu entdecken, kann jede angewendete Transformation verifiziert werden. Es ist aber auch möglich, die Korrektheit des Syntheseergebnisses durch den Nachweis der Äquivalenz von Spezifikation und Implementierung sicherzustellen. Abbildung 4.9 gibt einen Überblick.

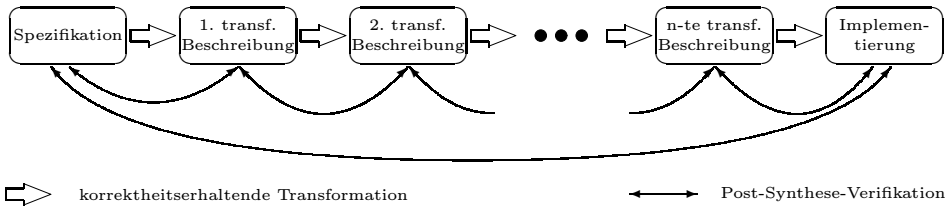


Abb. 4.9: Post-Synthese-Verifikation

Als Anwendungsbeispiel wird in Kapitel 5 gezeigt, wie unter Anwendung korrektheitserhaltender Transformationen aus einer sequentiellen Ausgangsbeschreibung einer Prozessorarchitektur ein Pipelinesystem synthetisiert werden kann. Die Verifikation des Prozessors mit Pipelining wird in Abschnitt 5.5 dargestellt. Die Äquivalenz der sequentiellen Spezifikation und des Prozessors mit Pipelining läßt sich durch ein Verfahrens, das auf Burch und Dill [BD94] zurückgeht, mit Hilfe einer symbolischen Simulation nachweisen.

4.10 Zusammenfassung

Das *TUD Transformationswerkzeug (TUDT)* führt korrektheitserhaltende Transformationen aus, die auf einem logischen Kalkül basieren. Der Kern des Werkzeugs besteht aus einer Menge von formalen Transformationen, die die Pfadäquivalenz einer Beschreibung erhalten und die den Kontrollfluß einer Beschreibung modifizieren, und einer Menge von kontext-abhängigen Transformationen, die korrektheitserhaltend im Sinne der Berechnungsäquivalenz sind und die die zeitlichen Eigenschaften einer Beschreibung verändern. Nach Anwenden einer Folge formaler und kontext-abhängiger Transformationen sind das Ergebnis und die ursprüngliche Beschreibung transformationsäquivalent.
Das Kalkül besteht aus 15 Axiomen und einer Regel. Die Vollständigkeit des Kalküls konnte nicht nachgewiesen werden, da es sich um Transformationen

handelt, die sowohl auf strukturelle als auch auf Verhaltensbeschreibungen angewendet werden. Die Anwendungsbeispiele legen dies jedoch nahe. Das Kalkül ist logisch korrekt; da jedoch aufgrund einer fehlerhaften Implementierung des Transformationswerkzeugs ein Ergebnis fehlerbehaftet sein kann, werden Verfahren vorgesehen, die die Korrektheit eines Ergebnisses durch eine unabhängige Post-Synthese-Verifikation nachweisen.

Kapitel 5

Synthese von Prozessoren mit Pipelining

5.1 Einleitung

Pipelining wird in modernen Prozessorarchitekturen eingesetzt, um den Befehlsdurchsatz zu erhöhen. Die Zeit für die Ausführung eines Befehls verkürzt sich nicht, sondern durch das parallele Bearbeiten von Instruktionen reduziert sich die Gesamtrechenzeit eines Programms. Diese Beschleunigung soll nicht durch zusätzliche Funktionseinheiten, sondern durch eine bessere Auslastung der vorhandenen Ressourcen erzielt werden. Die Befehle werden in mehrere Schritte bzw. Phasen unterteilt, um verschiedene Befehlsschritte aufeinanderfolgender Befehle zeitgleich ausführen zu können. Die Instruktionen werden in einem festen zeitlichen Abstand, der sogenannten *Latenzzeit*, gestartet.

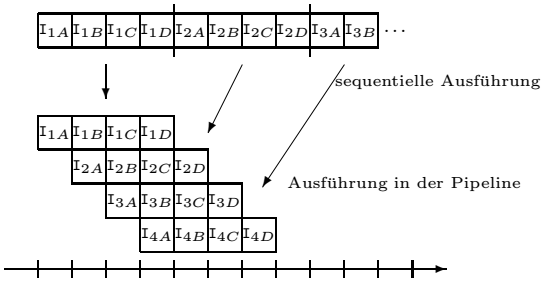


Abb. 5.1: Pipelining von vier Befehlen

Durch das um die Latenzzeit verzögerte Starten der Instruktionen wird vermieden, daß gleiche Befehlsphasen unterschiedlicher Befehle parallel ausgeführt werden. Zusätzliche Funktionseinheiten werden nur benötigt, wenn unterschiedliche Befehlsphasen gleiche Ressourcen verwenden. In [PK89a] wird diese Technik als *funktionales Pipelining* bezeichnet.

Beispiel 5.1 *Abbildung 5.1 stellt die sequentielle Ausführung von Befehlen der Ausführung in der Pipeline gegenüber. Die Befehle umfassen bis zu vier Phasen ($\mathtt{I}_A$ bis $\mathtt{I}_D$), wobei die Befehlsphasen, die nebeneinander stehen, sequentiell und die übereinander stehenden parallel ausgeführt werden. Die Latenzzeit der dargestellten Pipeline beträgt einen Zeitschritt, so daß bis zu vier Befehle gleichzeitig verarbeitet werden. Gleiche Befehlsphasen aufeinanderfolgender Befehle werden nie zeitgleich ausgeführt. So werden z.B. $\mathtt{I}_{1C}$ und $\mathtt{I}_{2C}$ in unterschiedlichen Zeitschritten realisiert.*

Abbildung 5.1 stellt eine "ideale" Pipeline dar. Da die Ausführung der Befehle nicht sequentiell erfolgt, sondern in der Pipeline überlappt wird, kann es zu Konflikten kommen. Diese entstehen durch Datenabhängigkeiten zwischen aufeinanderfolgenden Befehlen. Eine korrekte Einplanung bedeutet, daß alle Abhängigkeiten, die Konflikte verursachen, entdeckt und gelöst werden.
Shewa [PP88] stellt eines der ersten Systeme dar, das eine Pipeline-Synthese für einen Datenpfad durchführt. Es setzt voraus, daß zwischen Befehlen keine Datenabhängigkeiten bestehen. Konkurrieren Befehlsphasen um Funktionseinheiten, wird die Latenzzeit derart gewählt, daß diese Befehlsphasen nicht parallel ausgeführt werden. Als Ergebnis erzeugt Shewa eine *statische Pipeline*. Im Gegensatz zu *dynamischen Pipelines* werden in einer statischen Pipeline alle Operationen gleich eingeplant. Moderne Prozessoren mit Pipelining sind dynamisch, d.h. der Kontext, in dem ein Befehl ausgeführt wird, entscheidet über die Einplanung. So werden z.B. bei der Ausführung einer Instruktion nur Leerschritte eingefügt, wenn Abhängigkeiten zu einem vorangehenden Befehl bestehen.
[PK89a] schlägt eine Erweiterung des *Force Directed Scheduling (FDS)* Algorithmus vor, um eine Pipeline-Synthese vorzunehmen. FDS bestimmt für eine vorgegebene Verarbeitungszeit einen Ablaufplan, der minimale Ressourcen benötigt. Die Zuordnung der Operationen auf die Kontrollzustände erfolgt durch Minimieren einer Kostenfunktion, in die die Wahrscheinlichkeit eingeht, daß eine Operation in einem bestimmten Schritt auf einer bestimmten Funktionseinheit ausgeführt wird. Eine Pipeline wird eingeplant, indem eine Latenzzeit vorgegeben wird und die Operationen, die parallel auszuführen sind, dem gleichen Kontrollschritt zugeordnet werden. Die Einplanung dieser Kontrollschritte wird dann durch FDS vorgenommen. Wie in Shewa bleiben Datenabhängigkeiten von aufeinanderfolgenden Befehlen unbeachtet.
In [HHL91] dagegen werden Datenabhängigkeiten bei der Pipeline-Synthese berücksichtigt. Die Abhängigkeiten geben die Reihenfolge vor, in der die Operationen auszuführen sind. In der Pipeline bleibt diese Reihenfolge gewahrt. Techniken wie z.B. das *Einführen von Pipeline-Registern* und *Forwarding* bzw. *Bypassing* werden nicht eingesetzt, um Abhängigkeiten zwischen Operationen zu lösen. Als Ergebnis wird eine statische Pipeline erzeugt.
Die *Snapshot Methode* [CT93] plant dynamische Pipelines ein. Als *Snapshot* wird eine mögliche Folge von Befehlen, die in der Pipeline ausgeführt werden soll, be-

zeichnet. Unter Beachtung vorgegebener Ressourcenbeschränkungen werden die zu berücksichtigenden Snapshots sukzessive eingeplant. Da die Menge der Snapshots groß sein kann, werden die Befehle zu Instruktionsklassen zusammengefaßt. Die Anzahl der möglichen Permutationen kann dennoch extrem groß sein. Auftretende Datenabhängigkeiten werden durch das Einfügen von Leerschritten oder das Hinzufügen weiterer Funktionseinheiten gelöst. Andere wie z.B. die oben genannten Techniken zum Lösen von Abhängigkeiten werden nicht angewendet.
Die in diesem Kapitel dargestellte *fall-basierte Einplanung* [Hin98a, EHR98, HER99, HRE99] ist im Unterschied zu den bereits angesprochenen Techniken ein Verfahren der formalen Synthese. Korrektheitserhaltende Transformationen werden verwendet, um unter Ressourcenbeschränkung eine dynamische Pipeline einzuplanen. Da Fehler der Implementierung des Transformationswerkzeugs nicht auszuschließen sind, wird die Korrektheit der eingeplanten Pipeline durch einen unabhängigen, am Lehrstuhl entwickelten Äquivalenzprüfer nachgewiesen.
In einer Vorverarbeitung werden die Befehle in Befehlsphasen unterteilt. Es werden die Operationen festgelegt, die in der gleichen Befehlsphase auszuführen sind. Durch sukzessives Parallelisieren der Befehlsphasen wird die Pipeline gefüllt. Datenabhängigkeiten werden im Gegensatz zu den bereits erwähnten Ansätzen durch Techniken wie *Forwarding* bzw. *Bypassing* oder das *Einführen von Pipeline-Registern* gelöst, sofern die zur Verfügung stehenden Ressourcen dieses ermöglichen. Andernfalls werden Leerschritte eingefügt. Als Ergebnis wird eine Verhaltensbeschreibung einer Pipeline generiert, die sowohl das Füllen als auch das Leeren der Pipeline umfaßt.
Der Datenflußgraph ist nicht die geeignete Darstellungsform, um entscheiden zu können, ob Konflikte in der Pipeline zu lösen sind, da es nicht genügt, nur die Datenabhängigkeiten zu kennen. Im Gegensatz zu den herkömmlichen Einplanungsverfahren ermittelt das dargestellte Verfahren aus den Anweisungen keinen Datenflußgraphen, sondern arbeitet direkt auf deren interner Darstellung.
Während im nächsten Abschnitt das Verfahren zur *fall-basierten Einplanung* beschrieben wird, stellt Abschnitt 5.3 eine Erweiterung dar, die es erlaubt, Ressourcenbeschränkungen bei der Einplanung zu berücksichtigen. Das Erzeugen einer dreistufigen Pipeline, einer DLX mit Pipelining und eines PIC-Mikrocontroller dienen als Beispiel. Der Nachweis der Korrektheit der Einplanungsergebnisse und eine Zusammenfassung beschließen das Kapitel.

5.2 Erzeugen von Pipelinesystemen

5.2.1 Spezifikation des sequentiellen Prozessors

Die Spezifikation des sequentiellen Prozessors unterliegt bestimmten Vorgaben. Die Beschreibung darf lediglich aus einem Segment bestehen. Wie Abbildung

5.2 zu entnehmen ist, enthält das Segment eine if-then-else-Anweisung, deren else-Zweig die Befehle des Prozessors und deren then-Zweig das Verhalten nach Beendigung der Befehlsausführung beschreibt.

$\mathcal{B} = (\{L^0\}, L^0, ...)$ mit dem Segment

```
L^0 :
  IF <Bedingung für die Terminierung>
    THEN <Anweisungen nach der Befehlsabarbeitung>
    ELSE <Beschreibung der Befehle>
```

Abb. 5.2: Struktur der Spezifikation

Jeder Befehl wird in mehrere Befehlsphasen eingeteilt. Jede Befehlsphase kann in einem Zeitschritt abgearbeitet werden. Die erste Phase ist für alle Befehle gleich und beschreibt das Laden des nächsten Befehls. Danach wird entsprechend dem geladenen Befehl verzweigt. Daher besteht der else-Zweig der oben erwähnten if-then-else-Anweisung aus einer parallelen Anweisung, die das Laden des nächsten Befehls realisiert, und einer if-then-else-Anweisung, die dessen Ausführung beschreibt. Die Ausgangsmarke verweist auf den Beginn des Segments, so daß eine Schleife beschrieben wird. Die Anzahl der Befehle, die der Prozessor ausführt, und die Anzahl an Befehlsschritten, die die Instruktionen umfassen, ist dabei beliebig. Abbildung 5.3 gibt ein Beispiel.

$\mathcal{B} = (\{L^0\}, L^0, ...)$ mit dem Segment

```
L^0 :
  IF <Bedingung für die Terminierung>
    THEN <Anweisungen nach der Befehlsabarbeitung>
    ELSE <Laden eines Befehls>;
         IF <geladener Befehl ist ein load>
           <Spezifikation der zweiten Phase des load-Befehls>
           ...
           <Spezifikation der letzten Phase des load-Befehls>
         ELSIF <geladener Befehl ist ein store>
           <Spezifikation der zweiten Phase des store-Befehls>
           ...
           <Spezifikation der letzten Phase des store-Befehls>
         ELSIF ...
         L^0
```

Abb. 5.3: Spezifikation der Befehle

Beispiel 5.2 *Abbildung 5.4 stellt die Spezifikation eines vereinfachten DLX-Prozessors [HP96] dar. Der Befehlssatz der DLX wird in dem else-Zweig der if-then-else-Anweisung des Segments L^0 beschrieben. Das Laden des nächsten Befehls und das parallele Inkrementieren des Befehlszählers* `pc` *erfolgt durch die parallelen Zuweisungen* `(ir←imem[pc],pc←pc+1)`*. Die sich anschließende if-then-else-Anweisung verzweigt entsprechend des geladenen Befehls. In dem dargestellten Beispiel wird von fünf unterschiedlichen Befehlen ausgegangen, die Befehlsklassen repräsentieren.* `load`*,* `store`*,* `alu`*,* `branch` *und* `jump` *sind binäre*

Konstanten. Die Funktionen op1, op2, op3, ival *und* alu *abstrahieren von der Adressierung. Durch die Ausgangsmarke* L^0 *wird eine Schleife beschrieben.*

$\mathcal{B}^{DLX} = (\{L^0\}, L^0, \{ir, pc, a, b, ar, din, dout, temp, dmem, imem, rf\})$ mit dem Segment

```
L^0 :
  IF flush
    THEN stall;
         L^Ende;
    ELSE (ir←imem[pc],
         pc←pc+1);
         IF opcode(ir)=load
           THEN (a←rf[op1(ir)],
                b←rf[op2(ir)]);
                ar←a+ival(ir);
                din←dmem[ar];
                rf[op3(ir)]←din;
         ELSIF opcode(ir)=alu
           THEN (a←rf[op1(ir)],
                b←rf[op2(ir)]);
                temp←aluop(ir,a,b);
                rf[op3(ir)]←temp;
         ELSIF opcode(ir)=store
           THEN (a←rf[op1(ir)],
                b←rf[op2(ir)]);
                (ar←a+ival(ir),
                dout←b);
                dmem[ar]←dout;
         ELSIF (opcode(ir)=branch)∧
                     (rf[op1(ir)]=0)
           THEN pc←pc+ival(ir);
         ELSIF opcode(ir)=jump
           THEN pc←ival(ir);
         ELSE
           stall;
         L^0;
```

Abb. 5.4: Beispiel einer DLX

Der then-Zweig der if-then-else-Anweisung des Segments L^0 *beschreibt das Terminieren der Befehlsausführung. Da ein Beenden der Ausführung nur nach Abarbeiten eines Befehls möglich ist, führt der then-Zweig einen Leerschritt (*stall*) aus, um die Ausführung, signalisiert durch die Ausgangsmarke* L^{Ende}*, zu beenden.*

5.2.2 Ermitteln der Befehlsphasen

Die Spezifikation des sequentiellen Prozessors unterliegt, wie in Abschnitt 5.2.1 dargestellt, bestimmten Vorgaben. Der Befehlssatz des Prozessors ist *befehlsorientiert* beschrieben. Wie Abbildung 5.5(a) zu entnehmen ist, verzweigen verschachtelte if-then-else-Anweisungen entsprechend des geladenen Befehls. Jeder Zweig der if-then-else-Anweisungen beschreibt die Ausführung eines Befehls.
In Abbildung 5.5(b) wird die Spezifikation *phasenorientiert* dargestellt. Nach dem Laden eines Befehls folgen mehrere sequentielle if-then-else-Anweisungen, die jeweils die Ausführung einer Phase beschreiben.
Die befehlsorientierte Darstellung ist für den Benutzer übersichtlicher, während die phasenorientierte Darstellung dazu dient, ein Pipelinesystem zu erzeugen bzw. eine Einplanung durchzuführen. Durch eine Vorverarbeitung kann die befehlsorientierte Darstellung in eine phasenorientierte umgewandelt werden.

Beispiel 5.3 *Das Verfahren zur Umwandlung der befehls- in die phasenorientierte Darstellung wird anhand des Beispiels aus Abbildung 5.6 dargestellt. Vereinfachend wird davon ausgegangen, daß zwei Befehle, bestehend aus den Phasen* a1 *und* a2 *bzw.* b1 *und* b2*, ausgeführt werden.*
Durch Anwenden der Transformation A8 wird sowohl in dem then- als auch in

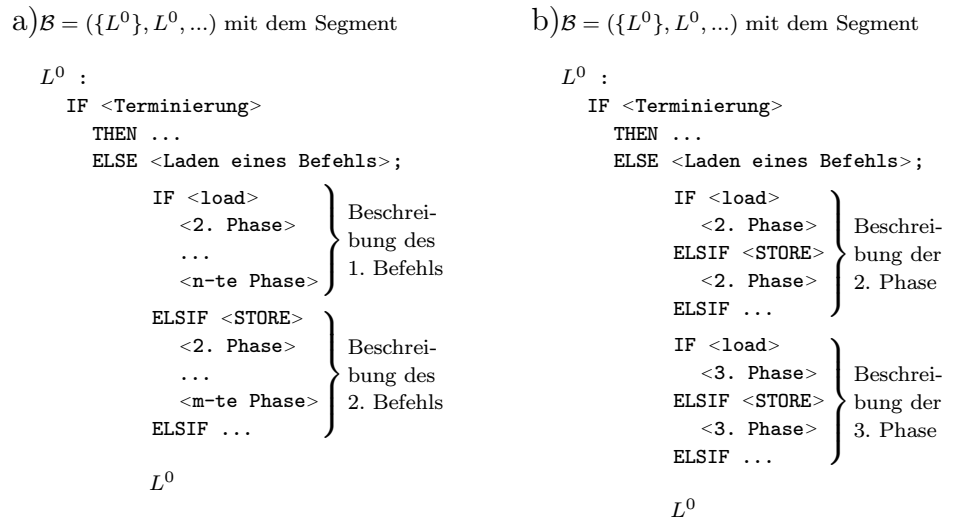


Abb. 5.5: Befehlsorientierte versus phasenorientierte Spezifikation

dem else-Zweig der if-then-else-Anweisung eine weitere if-then-else-Anweisung erzeugt. Aus den inneren if-then-else-Anweisungen kann mit Hilfe der Transformation A7 `a1` *bzw.* `b1` *herausgezogen werden.* `p` *repräsentiert eine Bedingung, die abfragt, welcher Befehl in das Instruktionsregister geladen wurde. Da das Instruktionsregister nur in der ersten Phase und nicht in* `a1` *bis* `b2` *beschrieben wird, gilt* $\{\texttt{a1}\}\texttt{p} \equiv \{\texttt{b1}\}\texttt{p} \equiv \texttt{p}$. *Die sowohl im then- als auch im else-Zweig enthaltenen if-then-else-Anweisungen gleichen sich, so daß durch Anwenden der Transformation A6 die if-then-else-Anweisungen herausgezogen werden können und sich die phasenorientierte Darstellung ergibt.*

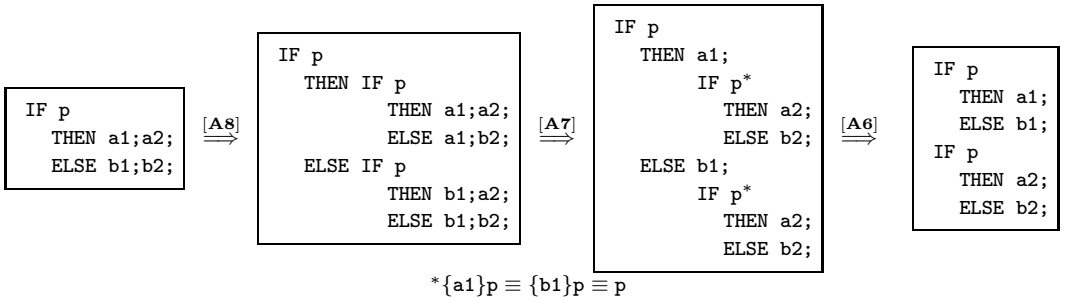


Abb. 5.6: Transformation der befehls- in die phasenorientierte Darstellung

Durch rekursives Anwenden dieses Verfahrens können auch Befehlssätze, die aus mehr als zwei Befehlsphasen bestehen, in die phasenorientierte Darstellung überführt werden.

Die Befehle, die ein Prozessor ausführt, bestehen aus unterschiedlich vielen Phasen, so daß die kürzeren Befehle durch Einfügen von Leerschritten (`stall`) verlängert werden müssen. Das Einfügen dieser Leerschritte hat einen entscheidenden Einfluß auf die benötigten Ressourcen.

Beispiel 5.4 *Die if-then-else-Anweisung in Abbildung 5.7(a) führt im then- zwei und im else-Zweig nur eine Anweisung aus. Es besteht die Möglichkeit, im else-Zweig einen Leerschritt nach* `c←ar+ival(ir)` *(Abbildung 5.7(b)) oder davor (Abbildung 5.7(c)) einzufügen. Wird 5.7(b) umgewandelt in die phasenorientierte Darstellung, greifen zwei verschiedene Phasen auf einen Addierer zu. Da im Pipelinesystem verschiedene Phasen parallel ausgeführt werden, konkurrieren diese Operationen um die vorhandenen Ressourcen. In 5.7(c) werden die Operationen in der gleichen Phase ausgeführt, so daß keine Ressourcenkonflikte bei der Ausführung in der Pipeline entstehen können.*

a)
```
IF p
    THEN (a←rf[op1(ir)],
          b←rf[op2(ir)]);
          c←a+b;
    ELSE c←ar+ival(ir);
```

b)
```
IF p
    THEN (a←rf[op1(ir)],
          b←rf[op2(ir)]);
          c←a+b;
    ELSE c←ar+ival(ir);
          stall;
```

c)
```
IF p
    THEN (a←rf[op1(ir)],
          b←rf[op2(ir)]);
          c←a+b;
    ELSE stall;
          c←ar+ival(ir);
```

Abb. 5.7: Einfügen von Leerschritten

Der längste Befehl gibt vor, in welchem Schritt welche Ressourcen verwendet werden. Die kürzeren werden entsprechend verlängert, so daß gleiche Ressourcen in gleichen Phasen angesprochen werden. Ist dies nicht möglich, können strukturelle Konflikte in der Pipeline auftreten, die für den Fall, daß keine zusätzlichen Funktionseinheiten zur Verfügung gestellt werden, durch das Anhalten der Pipeline zu lösen sind.

5.2.3 Die fall-basierte Einplanung

Die *fall-basierte Einplanung* erzeugt aus einer phasenorientierten Spezifikation ein System mit Pipelining. Wird die Spezifikation befehlsorientiert beschrieben, ist es notwendig, durch eine Vorverarbeitung eine phasenorientierte Darstellung zu erzeugen.

L^0 :
  IF <Bedingung für die Terminierung>
    THEN S_0^T; ...; S_m^T;
         L^{Ende};
    ELSE S_0^E; ...; S_n^E;
         L^0;

Abb. 5.8: Die phasenorientierte Spezifikation

Wie der Abbildung 5.8 zu entnehmen ist, besteht die Beschreibung aus einem Segment L^0, das eine if-then-else-Anweisung I_0 enthält. Der else-Zweig I_0^E umfaßt die sequentiellen Anweisungen S_0^E bis S_n^E, die den Befehlssatz phasenorientiert beschreiben. Die Ausgangsmarke L^0 verweist wieder auf den Beginn des

Segments. Der then-Zweig I_0^T beendet die Berechnung durch Ausführen der sequentiellen Anweisungen S_0^T bis S_m^T.
Der Algorithmus 5.1 leitet von einer sequentiellen Prozessorspezifikation eine Pipeline-Implementierung ab. $|I_x^{Zweig}|$ gibt die Anzahl der bedingten Anweisungen an. $S_{i=j}^{Zweig}$ meint die j-te Anweisung des jeweiligen if-then-else Zweiges, andernfalls sind die Sequenzen eindeutig indiziert.

Algorithmus 5.1 Fall-basierte Einplanung

input *das Anfangssegment* L_0

1. **let** $x = 0$;
2. **while** $|I_x^E| > 1$ **do**
3. *Erzeuge ein neues Segment* L_{x+1}*, das die Anweisungen* S_{x+1}^E *bis* S_n^E *von* I_x^E *aufnimmt, so daß nur* $S_{0\,bis\,x}^E$ *in* I_x^E *verbleibt* [**R1**];
4. *Ersetze die Ausgangsmarke* L_x *des Segments* L_{x+1} *durch den zugehörigen Segmentkörper, so daß die Folge der Anweisungen* S_{x+1}^E *bis* S_n^E *und der if-then-else-Anweisung* I_{x+1}*, einer Kopie von* I_x*, entsteht* [**R1**];
5. *Ziehe die Anweisungen* S_{x+1}^E *bis* S_n^E *in die if-then-else-Anweisung* I_{x+1} *hinein* [**A7**];
6. *Ziehe* $S_{0\,bis\,x}^E$*, die sich am Ende des else-Zweiges* I_{x+1}^E *befindet, vor die Anweisungen* S_{x+2}^E *bis* S_n^E *und parallelisiere* $S_{0\,bis\,x}^E$ *mit* S_{x+1}^E*, der ersten Anweisung des else-Zweiges* I_{x+1}^E *von* I_{x+1} [**C1-C3**];
7. **if** *es handelt sich nicht um die erste Iteration* **then**
8. *Parallelisiere paarweise die Anweisungen* $S_{i=0}^T$ *bis* $S_{i=n-x-1}^T$ *mit* S_{x+1}^E *bis* S_n^E [**C1-C3**];
9. **fi**;
10. **let** $x = x + 1$;
11. **od**;

return *die Segmente* L_0 *bis* L_n;

In Zeile 3 wird ein neues Segment L_{x+1} erzeugt, das die Anweisungen S_{x+1}^E bis S_n^E des else-Zweiges I_x^E der if-then-else-Anweisung I_x des Segments L^x aufnimmt. Die Anweisung $S_{0\,bis\,x}^E$, die in I_x^E verbleibt, beschreibt das parallele Ausführen von $x+1$ Befehlsphasen und beschreibt das Füllen der Pipeline. Sie repräsentiert die Anweisungen S_0^E bis S_x^E, die parallel ausgeführt werden. Im nächsten Schritt wird die Ausgangsmarke L^x des Segments L^{x+1} durch den Körper des zugehörigen Segments ersetzt (Zeile 4). Eine Kopie I_{x+1} der Anweisung I_x ersetzt L^x. Durch Hineinziehen der Anweisungen S_{x+1}^E bis S_n^E in die if-then-else-Anweisung I_{x+1} in Zeile 5 ist es möglich, S_{x+1}^E mit $S_{0\,bis\,x}^E$ zu parallelisieren. Die entstehende Anweisung $S_{0\,bis\,x+1}^E$ realisiert $x+2$ Befehlsphasen gleichzeitig.
Die Schleife (Zeilen 2-11) terminiert, da jede Iteration die Anzahl der Anweisungen in I_x^E um eins erniedrigt. Durch Ausführen der Schleife wird die Pipeline sukzessiv gefüllt. Bleibt eine Anweisung $S_{0\,bis\,n}^E$ in I_n^E zurück, so daß $|I_x^E| > 1$

nicht mehr erfüllt ist, terminiert der Algorithmus. Die Pipeline ist gefüllt bzw. der Pipelinezustand ist erreicht.
Die Anweisungen $S^T_{i=0}$ bis $S^T_{i=m}$ des then-Zweiges I^T_{x+1} beschreiben das Leeren der Pipeline für x geladene Befehle. In der Zeile 5 werden die Anweisungen S^E_{x+1} bis S^E_n in I^T_{x+1} hineingezogen. Durch das Hinzufügen von S^E_{x+1} bis S^E_n und paarweises Parallelisieren mit $S^T_{i=0}$ bis $S^T_{i=n-x-1}$ (Zeilen 7-9) wird das Leeren der Pipeline für $x+1$ geladene Befehle realisiert. Zu beachten ist, daß in der ersten Iteration (Zeilen 2-11) keine Anweisungen in Zeile 8 zu parallelisieren sind.

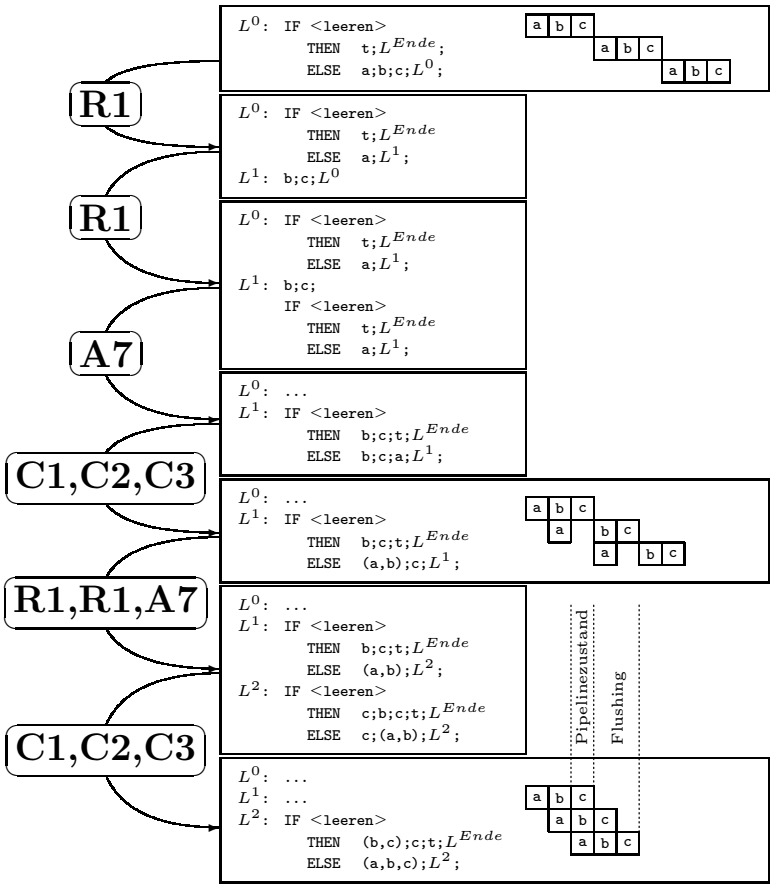


Abb. 5.9: Erzeugen einer dreistufigen Pipeline

Beispiel 5.5 *In Abbildung 5.9 wird unter Verwendung des Algorithmus 5.1 zur fall-basierten Einplanung eine dreistufige Pipeline erzeugt. Die Buchstaben* `a`*,* `b` *und* `c` *repräsentieren die jeweiligen Befehlsphasen. Ist die Bedingung der Verzweigung* <`leeren`> *erfüllt, beendet der Prozessor das Laden weiterer Befehle und führt die verbleibenden Befehlsphasen der bereits geladenen Instruktionen aus.*
Die erstmalige Ausführung der Regel R1 (Zeile 3) erzeugt ein neues Segment. Ein zweites Ausführen (Zeile 4) und das Hineinziehen von `b` *und* `c` *in die if-then-else-Anweisung entrollt die Schleife, die durch die Ausgangsmarke L^0 beschrieben wird. Durch Anwenden der Transformationen C1, C2 und C3 werden*

`a` *und* `b` *parallelisiert (Zeile 6).* `b` *repräsentiert die zweite Befehlsphase der zuerst geladenen Instruktion und* `a` *die erste Phase des nachfolgenden Befehls, so daß die Pipeline durch das Parallelisieren gefüllt wird. Es werden keine Anweisungen im then-Zweig der if-then-else-Anweisung parallelisiert, weil es sich um die erste Iteration des Algorithmus handelt (Zeilen 7-9). Zu diesem Zeitpunkt ist nur ein Befehl geladen, so daß bei einem Leeren nur ein Befehl terminiert werden muß. Durch Wiederholen der Transformationen entsteht ein neues Segment* L^2*. Die if-then-else-Anweisung realisiert in ihrem else-Zweig den Pipelinezustand, d.h.* `a`*,* `b` *und* `c` *werden parallel ausgeführt. Im then-Zweig werden* `c` *und* `b` *parallelisiert. Der then-Zweig beschreibt das Leeren der Pipeline für zwei geladene Befehle.*

5.3 Erzeugen von Pipelines unter Ressourcenbeschränkungen

5.3.1 Ressourcenbeschränkungen

Der in Abschnitt 5.2.3 vorgestellte Algorithmus zur fall-basierten Einplanung erzeugt aus der Spezifikation eines sequentiellen Prozessors die Beschreibung eines Prozessors mit Pipelining. Die Synthese wird unter Anwendung korrektheitserhaltender Transformationen durchgeführt. Da der Entwurf von Hardware meist den kritischen Umgang mit Ressourcen verlangt, muß dieser Aspekt während des Transformationsprozesses berücksichtigt werden.

Beispiel 5.6 *Die Zuweisungen* `pc←pc+1` *und* `c←a+b` *können parallelisiert werden, da keine Datenabhängigkeiten zwischen ihnen bestehen. Wird vorausgesetzt, daß nur ein Addierer zur Verfügung steht, ist es nicht möglich, die beiden Zuweisungen gleichzeitig auszuführen. Die Vorgabe eines Addierers erzwingt die sequentielle Abarbeitung der beiden Operationen und verbietet die Anwendung von Transformationen. Algorithmus 5.1 zur fall-basierten Einplanung berücksichtigt diese Beschränkung nicht.*

Um Beschränkungen der Ressourcen berücksichtigen zu können, ist das Verfahren zur fall-basierten Einplanung zu verfeinern. Der Benutzer kann die folgenden Einschränkungen spezifizieren:

- die Anzahl der funktionalen Einheiten und Speicher,
- die Zahl der parallelen Lese- und Schreibzugriffe auf die Speicher,
- die Verbindungen zwischen Registern und Speicher und
- die Latenzzeit.

5.3.2 Einplanung unter Ressourcenbeschränkungen

Das vorgestellte Verfahren der fall-basierten Einplanung gestattet es nicht, den Bedarf an Ressourcen einzuschränken. Möchte der Benutzer die Ressourcen vorgeben, ist es erforderlich, das Verfahren zu erweitern.
Wird durch das Parallelisieren zweier Befehlsphasen gegen die vorgegebenen Ressourcenbeschränkungen verstoßen, können diese nicht parallelisiert werden. Auch Datenabhängigkeiten, die nicht durch Forwarding oder Einführen von Pipeline-Registern gelöst werden können, machen es notwendig, die Pipeline anzuhalten. Da nicht alle Befehle in der gleichen Befehlsphase auf die gleichen Ressourcen zugreifen bzw. nicht jeder Befehl von einer vorausgehenden Instruktion abhängig ist, können die Befehlsphasen für bestimmte Befehle parallel ausgeführt werden. Für die übrigen Instruktionen müssen die Befehlsphasen teilweise sequentiell abgearbeitet werden; die Pipeline führt Leerschritte aus.

Beispiel 5.7 *In Abbildung 5.10 werden zwei Befehlsphasen beschrieben, die unter der Bedingung, daß nur ein Addierer zur Verfügung steht, nicht parallelisiert werden können. Die if-then-else-Anweisungen dekodieren den geladenen Befehl durch einen Vergleich mit den Konstanten* `jump`*,* `load` *und* `store`*. In der Dekodier-Phase verwendet nur der* `jump`*-Befehl einen Addierer, während die anderen Befehle auf Register der Registerbank zugreifen. Nur unter Ausschluß des* `jump`*-Befehls können die zwei Phasen parallel ausgeführt werden, wobei nur ein Addierer zum Einsatz kommt.*

Dekodier-Phase:

```
IF opcode(ir)=jump
  THEN pc←pc+ir[16:31];
  ELSE (a←rf[op1(ir)],
        b←rf[op2(ir)]);
```

Ausführungs-Phase:

```
IF opcode(p0)=load
  THEN ar←a+p0[16:31];
ELSIF opcode(p0)=store
  THEN (ar←a+p0[16:31],
        dout←b);
```

Abb. 5.10: Ressourcenkonflikt beim Parallelisieren zweier Befehlsphasen

Treten Konflikte auf, die nicht durch Forwarding oder Einführen von Pipeline-Registern gelöst werden können, wird eine weitere if-then-else-Anweisung eingeführt. Der then-Zweig beschreibt die Ausführung der Befehle, die ohne Verletzen der Ressourcenbeschränkungen parallel realisiert werden können. Der else-Zweig führt die übrigen Instruktionen teilweise sequentiell aus. Die Pipeline wird somit dynamisch eingeplant.
Wie Abbildung 5.11 zu entnehmen ist, wird auf die ursprüngliche, sequentielle Folge die Transformation A0 angewendet. Der then-Zweig $I^T_{Konflikt}$ und der else-Zweig $I^E_{Konflikt}$ der durch A0 eingeführten if-then-else-Anweisung $I_{Konflikt}$ sind identisch. Die Bedingung $C_{kein\,Konflik}$ wird derart bestimmt, daß keine Ressourcenkonflikte beim Parallelisieren der Anweisungen in $I^T_{Konflikt}$ möglich sind.

Abb. 5.11: Einführen einer weiteren if-then-else-Anweisung

Beispiel 5.8 *Die Befehlsphasen aus Abbildung 5.10 können unter Ausschluß des* `jump`*-Befehls parallelisiert werden. Durch Einführen einer if-then-else-Anweisung mit der Bedingung* `¬(opcode(ir)=jump)` *wird ausgeschlossen, daß ein Konflikt durch das Parallelisieren der Befehlsphasen auftreten kann. Sie tritt an die Stelle von $C_{kein\,Konflikt}$. Unter der Voraussetzung, daß* **keine Datenabhängigkeiten** *zwischen den beiden Befehlsphasen bestehen, ergeben sich die Anweisungen, die Abbildung 5.12 zu entnehmen sind.*

```
IF ¬(opcode(ir)=jump)
  THEN (a←rf[op1(ir)],
        b←rf[op2(ir)],
        IF opcode(p0)=load
          THEN ar←a+p0[16:31]
        ELSIF opcode(p0)=store
          THEN ar←a+p0[16:31],
               dout←b);
  ELSE pc←pc+ir[16:31];
        IF opcode(p0)=load
          THEN ar←a+p0[16:31];
        ELSIF opcode(p0)=store
          THEN (ar←a+p0[16:31],
               dout←b);
```

Abb. 5.12: Parallelisieren bei Einhaltung der Ressourcenbeschränkungen

Der then-Zweig führt die Befehlsphasen parallel aus. Da vorausgesetzt wird, daß keine Datenabhängigkeiten bestehen, wird kein Forwarding für die Register `a` *und* `b` *durchgeführt. Der else-Zweig führt* `pc←pc+ir[16:31]` *und die Ausführungs-Phase sequentiell aus. Die Pipeline wird somit für einen Verarbeitungsschritt angehalten bzw. ein Pipelinestall wird durchgeführt.*

Durch Einführen der Bedingung $C_{kein\,Konflik}$ entstehen sowohl im then- als auch im else-Zweig logisch unmögliche Pfade. Unter Verwendung binärer Entscheidungsgraphen [Bry86] werden logisch unmögliche Pfade entdeckt und eliminiert, indem die if-then-else-Anweisungen, die zu entscheiden sind, durch den jeweils ausgewählten Zweig ersetzt werden. Ist es nicht möglich, die if-then-else-Anweisungen zu entscheiden, werden ihre Bedingungen, die während der Synthese sehr komplex werden können, vereinfacht. Die Techniken, die es erlauben, algorithmische Beschreibungen zu vereinfachen, werden in Kapitel 7 dargestellt. Ohne das Vereinfachen der Beschreibungen ist eine automatisierte Pipeline-Synthese bei komplexeren Problemen nicht mehr möglich.

Beispiel 5.9 *Abbildung 5.13(a) stellt die if-then-else-Anweisung $I_{Konflikt}$ dar, die sich nach Ausführen der Transformation A0 ergibt. Die ausgewählte Bedingung* ¬(opcode(ir)=jump) *gestattet es, sowohl den then- als auch den else-Zweig der if-then-else-Anweisung zu vereinfachen. Abbildung 5.13(b) ist die vereinfachte Beschreibung zu entnehmen.*

a)
```
IF ¬(opcode(ir)=jump)
    THEN IF opcode(ir)=jump
           THEN pc←pc+ir[16:31];
           ELSE (a←rf[op1(ir)],
                 b←rf[op2(ir)]);
         IF opcode(p0)=load
           THEN ar←a+p0[16:31];
         ELSIF opcode(p0)=store
           THEN (ar←a+p0[16:31],
                 dout←b);
    THEN IF opcode(ir)=jump
           THEN pc←pc+ir[16:31];
           ELSE (a←rf[op1(ir)],
                 b←rf[op2(ir)]);
         IF opcode(p0)=load
           THEN ar←a+p0[16:31];
         ELSIF opcode(p0)=store
           THEN (ar←a+p0[16:31],
                 dout←b);
```

b)
```
IF ¬(opcode(ir)=jump)
    THEN (a←rf[op1(ir)],
          b←rf[op2(ir)]);
          IF opcode(p0)=load
            THEN ar←a+p0[16:31];
          ELSIF opcode(p0)=store
            THEN (ar←a+p0[16:31],
                  dout←b);
    THEN pc←pc+ir[16:31];
          IF opcode(p0)=load
            THEN ar←a+p0[16:31];
          ELSIF opcode(p0)=store
            THEN (ar←a+p0[16:31],
                  dout←b);
```

Abb. 5.13: Vereinfachen von $I_{Konflikt}$

Die if-then-else-Anweisung der Dekodier-Phase enthält logisch unmögliche Pfade, da sich die Bedingungen von $I_{Konflikt}$ und opcode(ir)=jump *widersprechen. Im then-Zweig von $I_{Konlikt}$ ersetzt* (a←rf[op1(ir)],b←rf[op2(ir)]) *und im else-Zweig* pc←pc+ir[16:31] *die if-then-else-Anweisung.*

Die Schritte 6 und 8 des Verfahrens zur fall-basierten Einplanung aus Abschnitt 5.2.3 werden durch die Funktionsaufrufe *Parallelize(I_x^T)* bzw. *Parallelize(I_x^E)* ersetzt. Algorithmus 5.2 *Parallelize* parallelisiert die entsprechenden Anweisungen, falls durch das parallele Ausführen der Anweisungen keine Ressourcenbeschränkungen verletzt werden. Die Funktion *Find-Violations(S_1 bis S_n)* (siehe Abschnitt 5.3.3) bestimmt die Verletzungen der Ressourcenbeschränkungen.
Andernfalls wird entsprechend dem oben geschilderten Verfahren eine neue if-then-else-Anweisung eingeführt. $C_{kein\,Konflikt}$ wird durch die Funktion *Find-Condition($I_{Konflikt}$)* (siehe Abschnitt 5.3.4) bestimmt. Die Anweisungen des then-Zweiges können ohne Verletzen der Beschränkungen parallelisiert werden.
Die Anweisungen des else-Zweiges werden durch die Funktion *Parallelize-Feasible($I^E_{Konflikt}$)* parallelisiert. Nur diejenigen Befehlsphasen werden parallel ausgeführt, die keine der Ressourcenbeschränkungen verletzen. Andernfalls werden Leerschritte eingefügt.

Algorithmus 5.2 Parallelize

input *ein Zweig I^{Zweig} einer if-then-else-Anweisung I;*

1. **if** I^{Zweig} *besteht aus einer if-then-else-Anweisung* I_x **then**
2. *Parallelize*(I_x^T);
3. *Parallelize*(I_x^E);
4. **elsif** *Find-Violations*(S_0^{Zweig} *bis* S_k^{Zweig})$\neq \emptyset$ **then**
5. *Erzeuge eine neue if-then-else-Anweisung* $I_{Konflikt}$ *und kopiere* S_0^{Zweig} *bis* S_k^{Zweig} *sowohl in den then-* $I_{Konflikt}^T$ *als auch in den else-Zweig* $I_{Konflikt}^E$ *ein* [**A0**];
6. *Bestimme* $C_{kein\,Konflikt}$ *durch Find-Condition*($I_{Konflikt}$)*, so daß die Anweisungen des then-Zweiges* $I_{Konflikt}^T$ *ohne Verletzen der Beschränkungen parallelisiert werden können;*
7. *Parallelize*($I_{Konflikt}^T$);
8. *Parallelize-Feasible*($I_{Konflikt}^E$);
9. **elsif** $\begin{pmatrix} \textit{handelt es sich bei } I^{Zweig} \textit{ um den Pipelinezustand oder füllt } I^{Zweig} \textit{ die} \\ \textit{Pipeline} \end{pmatrix}$ **then**
10. *Ziehe die letzte Anweisung* S_k^{Zweig} *vor* S_1^{Zweig} *bis* S_{k-1}^{Zweig} *und parallelisiere sie mit* S_0^{Zweig} [**C1-C3**];
11. **else**
12. *Parallelisiere die Anweisungen* S_0^{Zweig} *bis* S_s^{Zweig} *mit* S_{s+1}^{Zweig} *bis* S_k^{Zweig} [**C1-C3**];
13. **if** $k - s < s + 1$ **then**
14. *Führe die fall-basierte Einplanung für* I^{Zweig} *aus;*
15. **fi**;
16. **fi**;

return *die parallelisierten Anweisungen;*

Verletzt das Parallelisieren von sequentiellen Anweisungen Ressourcenbeschränkungen, wird entsprechend der geschilderten Technik durch die Zeilen 5 bis 8 eine weitere if-then-else-Anweisung eingeführt. Andernfalls werden in Zeile 10 die Anweisungen, die das Füllen der Pipeline beschreiben, parallelisiert. Die Zeilen 12 bis 15 parallelisieren die Anweisungen, die das Leeren der Pipeline beschreiben.

Beispiel 5.10 *In Abbildung 5.14 wird vorausgesetzt, daß die Befehlsphasen* `a`*,* `b` *und* `c` *für bestimmte Befehle unter Einhalten der vorgegebenen Ressourcen nicht parallel ausgeführt werden können. Aus diesem Grund wird eine weitere if-then-else-Anweisung eingeführt. Sie besitzt einen identischen then- und else-Zweig.* `¬Konflikt` *repräsentiert eine Bedingung, die jede Verletzung der Ressourcenvorgaben bei parallelem Ausführen von* `a`*,* `b` *und* `c` *ausschließt, so daß die Anweisungen des then-Zweiges parallelisiert werden können. Im else-Zweig wird das Auftreten von Konflikten durch Einfügen eines Leerschrittes vermieden.*

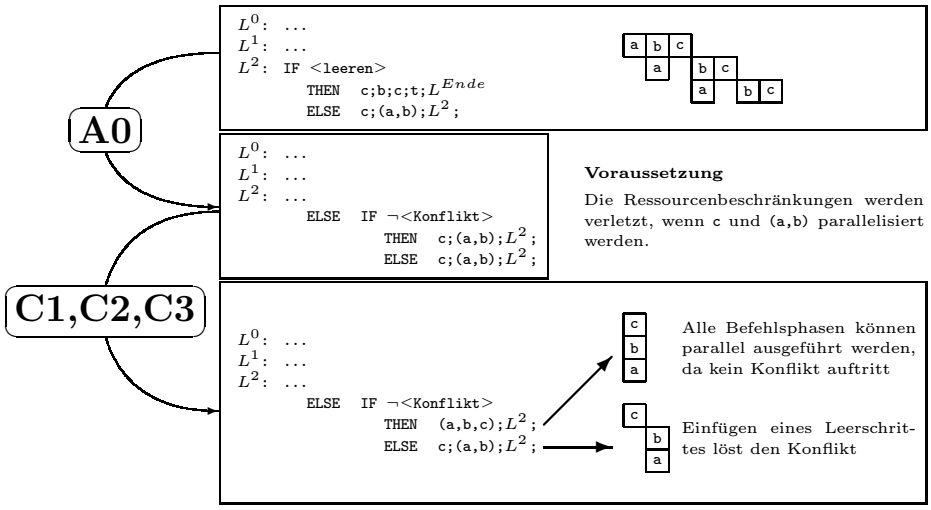


Abb. 5.14: Lösen von Konflikten während des Parallelisierens

Ist $k - s < s + 1$ in Zeile 13 erfüllt, wird die fall-basierte Einplanung nach dem Parallelisieren erneut gestartet. Da Instruktionen aus unterschiedlich vielen Schritten bestehen können, ist es möglich, daß Befehle bereits abgearbeitet wurden, bevor die vorausgehenden ausgeführt worden sind. In diesem Fall ist $k - s < s + 1$ erfüllt und die verbleibenden Anweisungen werden in das nächste Segment verschoben, um parallel zu den nachfolgenden Befehlen realisiert zu werden.

a)
```
ar←a+ival(ir);      } die Phasen des load
din←dmem[ar];       }
rf[op3(ir)]←din;    }
pc←pc+p0[16:31];    } die Phase des jump
L^x;
```

b)
```
(ar←a+ival(ir),     } die 1. Phase von
pc←pc+p0[16:31]);   } load und jump
din←dmem[ar];       } die übrigen Phasen
rf[op3(ir)]←din;    } des load
L^x;
```

c)
```
(ar←a+ival(ir),
pc←pc+p0[16:31]);
L^y;
```

Abb. 5.15: Rekursiver Aufruf der fall-basierten Einplanung

Beispiel 5.11 *In dem Beispiel in Abbildung 5.15(a) müssen drei Befehlsphasen des* `load`*-Befehls gefolgt von einer Phase des* `jump`*-Befehls ausgeführt werden. Abbildung 5.15(b) ist das Ergebnis der Einplanung zu entnehmen.* `pc←pc+p0[16:31]` *wird parallel zu* `ar←a+ival(ir)` *ausgeführt, so daß der* `jump`*-Befehl bereits terminiert. Um den vorangehenden* `load`*-Befehl zu beenden, müssen noch zwei weitere Phasen realisiert werden. Das nachfolgende Segment L^x beschreibt die Ausführung der nachfolgenden Befehle. Die sequentielle Ausführung der beiden letzten Phasen des* `load`*-Befehls läßt sich vermeiden, indem das Verfahren zur fall-basierten Einplanung rekursiv auf diese beiden Phasen angewendet wird. Durch Anwendung der Regel R1 wird ein neues Segment L^y erzeugt. Die Anweisungen* `din←dmem[ar]` *und* `rf[op3(ir)]←din` *sowie die Ausgangsmarke L^x*

werden durch die Marke des neuen Segments ersetzt, dessen Segmentkörper sie bilden. Wie in 5.15(c) zu sehen ist, bleiben die Anweisungen `pc←pc+p0[16:31]` *und* `ar←a+ival(ir)` *unverändert.*
Die Einplanung des neuen Segments wird durch rekursives Aufrufen des Algorithmus der fall-basierten Einplanung durchgeführt. Das entstehende Segment führt `din←dmem[ar]` *und* `rf[op3(ir)]←din` *parallel zu den nachfolgenden Befehlen aus. In diesem Beispiel gilt* $s = 2$ *und* $k = 3$*, so daß* $k - s < s + 1$ *(Zeile 13) erfüllt ist.*

Die Funktion *Parallelize-Feasible*($I^E_{Konflikt}$) parallelisiert jene Anweisungen, bei deren Abarbeitung die Pipeline angehalten werden muß.

Algorithmus 5.3 Parallelize-Feasible

input *die Anweisungen* S^E_0*; ...;* S^E_s*;* S^E_{s+1}*; ...;* S^E_k *des else-Zweiges* I^E_x *der if-then-else-Anweisung* I_x*;*

1. **let** $k_1 = 0$ *und* $k_2 = s + 1$*;*
2. **while** $k_1 < s + 1$ **do**
3. **if** *Find-Violations*($S^E_{k_1}$ *und* $S^E_{k_2}$)$\neq \emptyset$ **then**
4. **let** $k_1 = k_1 + 1$*;*
5. **else**
6. *Parallelisiere* $S^E_{k_1}$ *und* $S^E_{k_2}$ **[C1-C3]***;*
7. **let** $k_1 = k_1 + 1$ *und* $k_2 = k_2 + 1$*;*
8. **fi***;*
9. **od***;*
10. **if** $k - s < s + 1$ **then**
11. *Führe die fall-basierte Einplanung für* I^E_x *aus;*
12. **fi***;*

return *die parallelisierten Anweisungen;*

Zeile 3 prüft, ob die Anweisungen $S^E_{k_1}$ und $S^E_{k_2}$ parallelisiert werden können. Treten durch das gleichzeitige Ausführen der beiden Anweisungen Verletzungen der Vorgaben ein, wird k_1 in Zeile 7 inkrementiert. Andernfalls werden $S^E_{k_1}$ und $S^E_{k_2}$ in Zeile 6 parallelisiert. Die Zeilen 10 bis 12 rufen, analog zu den Zeilen 13 bis 15 des Algorithmus *Parallelize*, die fall-basierte Einplanung rekursiv auf.

5.3.3 Verletzungen der Ressourcenvorgaben

Ein Großteil der Berechnungszeit wird dazu verwendet, die Verletzungen der Ressourcenvorgaben zu ermitteln. Einige Beschränkungen lassen sich sehr leicht überprüfen. So kann z.B. schnell kontrolliert werden, ob die Speicher mit den vorgegebenen Registern verbunden sind. Hingegen quantitative Vorgaben zu überprüfen, ist sehr schwierig, da Verhaltensbeschreibungen betrachtet werden. So ist es z.B. aufwendig, Beschränkungen wie die Anzahl der funktionalen Einheiten oder der parallelen Lese- und Schreibzugriffe auf die Speicher zu überprüfen. Die Ausführung paralleler Anweisungen ist abhängig von komplexen Bedingungen. Um Verletzungen der Vorgaben zu entdecken, muß jede mögliche Kombination der parallel auszuführenden Anweisungen betrachtet werden.

Beispiel 5.12 *Abbildung 5.16(a) stellt zwei verschachtelte if-then-else-Anweisungen dar, die unter der Voraussetzung, daß nur ein Addierer verwendet wird, parallelisiert werden sollen. Die Anweisungen können nicht in jedem Fall gleichzeitig ausgeführt werden. Gilt* $\neg p \wedge q = 1$*, wären zwei Addierer erforderlich, um* `a←x+z` *und* `b←y+z` *parallel zu berechnen. In den übrigen Fällen besitzen die Zuweisungen zwar unterschiedliche Ziele, führen aber die gleichen Operationen aus, so daß ein Addierer ausreicht, die Berechnungen zu realisieren.*
Abbildung 5.16(b) zeigt die parallelisierten Anweisungen. Ein Addierer genügt, um die Berechnungen auszuführen. Da für $\neg p \wedge q = 1$ *die Beschränkungen durch Parallelisieren verletzt werden, erfordert die Ausführung der Zuweisungen zwei Zeitschritte. In den übrigen Fällen können die Zuweisungen in einem Zeitschritt unter Einhaltung der Ressourcenbeschränkungen realisiert werden.*

a)
```
IF p                          IF p
  THEN   a←x+y;                 THEN   b←x+y;
  ELSIF q          und          ELSIF q
    THEN a←x+z;                   THEN b←y+z;
    ELSE a←x;                     ELSE b←x+z;
```

b)
```
IF p
  THEN   (a←x+y,b←x+y);
  ELSIF q
    THEN a←x+z;b←y+z;
    ELSE (a←x,b←x+z);
```

Abb. 5.16: Verletzung von Ressourcenbeschränkungen

Um Verletzungen der Ressourcenvorgaben zu entdecken, werden die betreffenden Anweisungen parallelisiert. Anschließend werden die entstehenden Anweisungen auf Verstöße gegen die Vorgaben überprüft. Verletzungen werden entdeckt, indem die Anweisungen symbolisch simuliert werden. Teilausdrücke der Bedingungen, sogenannte *atomare Formeln* (siehe auch Abschnitt 7.4), werden sukzessive als *wahr* oder $falsch$ entschieden, so daß sich die Wahrheitswerte aller auftretenden Bedingungen bestimmen lassen. Somit ergibt sich für eine bestimmte Kombination von erfüllten und nicht erfüllten Bedingungen eine Folge von Zuweisungen, die leicht auf Verletzungen der Vorgaben überprüft werden kann. Jede Kombination von *wahren* und $falschen$ *atomaren Formeln* wird rekursiv simuliert. Als

atomare Formeln werden alle Formeln, mit Ausnahme boolescher Variablen und der Operationen $\wedge$, $\vee$ und $\neg$, bezeichnet, aus denen die Bedingungen zusammengesetzt sind.

```
IF (opcode(ir)=load)∧¬(opcode(p0)=store)
  THEN   (a←x+y,b←x+y);
  ELSIF (opcode(ir)=load)∨(opcode(ir)=add)
    THEN (a←x+z,b←y+z);
    ELSE (a←x,b←x+z);
```

Abb. 5.17: Symbolische Simulation der parallelisierten Anweisungen

Beispiel 5.13 *Repräsentiert in Abbildung 5.17 p die Bedingung* `(opcode(ir)=load)∧¬(opcode(p0)=store)` *und q* `(opcode(ir)=load)∨(opcode(ir)=add)`*, dann werden alle Kombinationen von Wahrheitswerten der Formeln* `opcode(ir)=load`*,* `opcode(ir)=add` *und* `opcode(p0)=store` *gebildet.* `load`*,* `add` *und* `store` *repräsentieren binäre Konstanten. In dem Fall, daß* `opcode(ir)=load` *als wahr und* `opcode(p0)=store` *als falsch angenommen wird, führt die if-then-else-Struktur* `(a←x+y,b←x+y)` *aus. Entsprechend ergibt sich in dem Fall, daß* `opcode(ir)=load` *und* `opcode(p0)=store` *als wahr entschieden werden,* `(a←x+z,b←y+z)`*. Während im ersten Fall die Beschränkung, daß nur ein Addierer zur Verfügung steht, eingehalten wird, sind im zweiten Fall für die parallele Ausführung der Berechnung zwei Addierer erforderlich.*

Algorithmus 5.4 Find-Violations

input *die parallelen Anweisungen T_1,...,T_n und die Menge $\mathcal{C}$ aller atomarer Formeln, die in T_1,...,T_n zu finden sind;*

1. **if** $\mathcal{C} = \emptyset$ **then**
2. **if** *in T_1,...,T_n werden alle Beschränkungen eingehalten* **then**
3. **return** $\emptyset$*;*
4. **else**
5. **return** *die Menge der verletzten Ressourcenbeschränkungen;*
6. **fi***;*
7. **else**
8. **let** $x \in \mathcal{C}$*;*
9. **return** $\left(\begin{array}{l} \textit{Find-Violations}(T_1,...,T_n,\mathcal{C})|_{x=TRUE} \bigcup \\ \textit{Find-Violations}(T_1,...,T_n,\mathcal{C})|_{x=FALSE} \end{array} \right)$*;*
10. **fi***;*

return $\emptyset$ *oder die verletzten Ressourcenbeschränkungen*

Die Funktion *Find-Violations* ermittelt durch symbolische Simulation für eine Folge paralleler Anweisungen T_1 bis T_n alle Verstöße gegen die Ressourcenvorgaben. Die Menge $\mathcal{C}$ enthält alle atomaren Formeln, die in den Bedingungen der Anweisungen T_1 bis T_n auftreten. In Zeile 8 wird sukzessive eine Bedingung x ausgewählt, der in Zeile 9 ein Wahrheitswert zugeordnet wird, bevor *Find-Violations* rekursiv aufgerufen wird. Indem x als wahr oder falsch angenommen wird, vereinfachen sich einerseits die Anweisungen T_1 bis T_n, andererseits verkleinert sich die Menge $\mathcal{C}$. Durch Ausnutzen des wechselseitigen Ausschlusses (siehe Abschnitt 7.4.4) und der Implikation von atomaren Formeln (siehe Abschnitt 7.4.5) können oft mehrere Elemente aus $\mathcal{C}$ ausgeschlossen werden.

Beispiel 5.14 *In Abbildung 5.17 treten die Bedingungen* `opcode(ir)=add`*,* `opcode(ir)=load` *und* `opcode(p0)=store` *auf. Nicht jede Kombination der Wahrheitswerte wird gebildet, da sich* `opcode(ir)=load` *und* `opcode(ir)=add` *unter der Voraussetzung, daß* `load` *und* `add` *unterschiedliche Konstanten repräsentieren, gegenseitig ausschließen. Nur eine dieser beiden atomaren Formeln kann als wahr angenommen werden, so daß sich nur sechs Kombinationen ergeben.*

Da $\mathcal{C}$ bei jedem Aufruf um mindestens ein Element reduziert wird, terminiert die Funktion. Sind allen atomaren Formeln in C Wahrheitswerte zugeordnet (Zeile 1), ergibt sich eine Folge von Zuweisungen. T_1 bis T_n enthalten keine Verzweigungen mehr, so daß etwaige Verletzungen der Ressourcenvorgaben ermittelt werden können (Zeilen 2-6).

5.3.4 Bedingungen für Konflikte in der Pipeline

Verletzungen der Ressourcenvorgaben werden gelöst, indem eine neue if-then-else-Anweisung $I_{Konflikt}$ eingeführt wird. Der then-Zweig $I^T_{Konflikt}$ führt jene Anweisungen aus, die ohne Verstöße gegen die Vorgaben parallelisiert werden können. Die Bedingung $C_{kein\,Konflikt}$ von $I_{Konflikt}$ wird derart bestimmt, daß sie alle Fälle ausschließt, in denen beim Parallelisieren Konflikte auftreten.
Um die Bedingung $C_{kein\,Konflikt}$ bestimmen zu können, werden die sequentiellen Anweisungen parallelisiert. Im Anschluß werden alle Bedingungen, die eine Verletzung der Ressourcenvorgaben implizieren, herausgesucht, negiert und durch Konjunktion miteinander verknüpft. Solange $C_{kein\,Konflikt}$ gültig ist, treten bei der Ausführung der parallelisierten Anweisungen keine Konflikte auf. Alle jene Pfade, auf denen Konflikte auftreten, werden durch $C_{kein\,Konflikt}$ logisch unmöglich. Die in Kapitel 7 dargestellten Techniken erlauben es, die logisch unmöglichen Pfade zu entdecken und zu eliminieren.

Beispiel 5.15 *In Abbildung 5.18 werden zwei Anweisungen unter der Voraussetzung parallelisiert, daß nur ein Addierer zur Verfügung steht.* `load` *und* `add`

repräsentieren binäre Konstanten. Ist eine der Bedingungen `opcode(p2)=load` *oder* `opcode(p2)=add` *erfüllt, werden jedoch zwei Addierer benötigt, da parallel zu der if-then-else-Anweisung die Zuweisung* `pc←pc+1` *ausgeführt wird.* `¬(opcode(p2)=load)∧¬(opcode(p2)=add)` *verhindert, daß Ressourcenverletzungen im then-Zweig* $I^T_{Konflikt}$ *der if-then-else-Anweisung* $I_{Konflikt}$ *auftreten. Der else-Zweig* $I^E_{Konflikt}$ *realisiert die Operationen, die durch einen* `load-` *bzw.* `add-`*Befehl ausgeführt werden. Durch Einführen der Bedingung* `¬(opcode(p2)=load) ∧¬(opcode(p2)=add)` *werden sowohl in dem then- als auch in dem else-Zweig Anweisungen logisch unmöglich.*

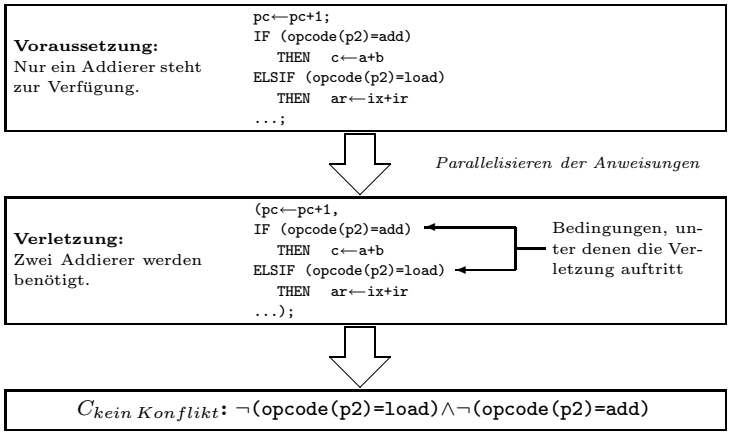


Abb. 5.18: Erzeugen der Bedingung $C_{kein\ Konflikt}$

Die Funktion *Find-Condition* ermittelt die Bedingung $C_{kein\,Konflikt}$. Die sequentiellen Anweisungen S^T_0 bis S^T_k des then-Zweiges $I^T_{Konflikt}$ der if-then-else-Anweisung $I_{Konflikt}$ sind bereits parallelisiert.

Algorithmus 5.5 Find-Condition

input *die parallelisierten Anweisungen* S^T_0 *bis* S^T_k *des then-Zweiges der if-then-else-Anweisung* $I_{Konflikt}$*;*

1. **let** $C_{kein\,Konflikt} = True$*;*
2. **foreach** $KONFLIKT \in \left\{ \begin{matrix} \textit{Konflikte, die während des Parallelisierens der Anwei-} \\ \textit{sungen in } I^T_{Konflikt} \textit{ auftreten} \end{matrix} \right\}$ **do**
3. **let** $\mathcal{C}$ = *alle Bedingungen der Anweisungen* S^T_0 *bis* S^T_k*, die* $KONFLIKT$ *bedingen;*
4. **while** $\left(\begin{matrix} \textit{das Parallelisieren von } S^T_0 \textit{ bis } S^T_k \textit{ unter } C_{kein\,Konflikt} \\ \textit{verletzt die Vorgaben} \end{matrix} \right) \bigwedge (\mathcal{C} \neq \emptyset)$ **do**
5. **let** $C_{kein\,Konflikt} = \begin{cases} (C_{kein\,Konflikt} \backslash r) \bigwedge \neg s & :\exists s \in \mathcal{C} : \exists r \in C_{kein\,Konflikt} : \\ & (s \bigwedge r = 1) \bigvee (s \bigwedge r = 0); \\ C_{kein\,Konflikt} \bigwedge \neg s & :\exists s \in \mathcal{C}; \end{cases}$
6. **let** $C = \begin{cases} (C \backslash s) \bigwedge r & :\exists s \in \mathcal{C} : \exists r \in C_{kein\,Konflikt} : (s \bigwedge r = 1) \bigvee (s \bigwedge r = 0); \\ C \backslash s & :\exists s \in \mathcal{C}; \end{cases}$
7. **od***;*

8. **od**;

return $C_{kein\,Konflikt}$;

Da mehrere Konflikte durch das Parallelisieren auftreten können, beschreiben die Zeilen 2 bis 8 eine Schleife. In Zeile 3 wird für jeden Konflikt die Menge aller Bedingungen $\mathcal{C}$ gebildet, die den Konflikt implizieren. Sukzessiv wird eine Bedingung aus $\mathcal{C}$ ausgewählt, negiert und mit $C_{kein\,Konflikt}$ konjunktiv verknüpft (Zeilen 4-7). Nicht alle Bedingungen aus $\mathcal{C}$ werden ausgewählt, um $C_{kein\,Konflikt}$ zu bilden, weil einige Bedingungen sich gegenseitig implizieren. Durch die Fallunterscheidung in Zeile 5 wird vermieden, daß widersprüchliche Bedingungen in $C_{kein\,Konflikt}$ aufgenommen werden.
S_0^T bis S_k^T repräsentieren eine Folge sequentiell auszuführender Anweisungen. Wird eine Bedingung ausgewählt, die sequentiell abhängig von vorausgehenden Anweisungen ist, müssen die Datenabhängigkeiten durch Forwarding (siehe Abschnitte 4.7 und 7.5) gelöst werden.

Beispiel 5.16 *In Abbildung 5.19(a) tritt ein Ressourcenkonflikt unter der Voraussetzung auf, daß nur ein Addierer zur Verfügung steht. Um die Zuweisungen* `pc←pc+1` *und* `c←a+b` *parallel ausführen zu können, werden aber zwei Addierer benötigt. Die Bedingung* $C_{kein\,Konflikt}$ *muß derart bestimmt werden, daß die Zuweisung* `c←a+b` *nicht im then-Zweig, sondern nur im else-Zweig realisiert wird.* `add` *repräsentiert eine binäre Konstante.*
`opcode(p2)=add` *ist die Bedingung, unter der der Ressourcenkonflikt auftritt und die in Abbildung 5.19(b) als* $C_{kein\,Konflikt}$ *dient. Die Datenabhängigkeiten, die zwischen* `opcode(p2)=add` *und den vorausgehenden Zuweisungen* `p2←p1` *und* `p1←ir` *bestehen, bleiben dabei unberücksichtigt. Aus diesem Grund verhindert die Bedingung* `¬(opcode(p2)=add)` *in 5.19(b) nicht das Auftreten des Konflikts.*

a)
```
IF C_kein Konflikt
   THEN (ir←imem[pc],
         p1←ir);
         p2←p1;
         (pc←pc+1,
         IF opcode(p2)=add
           THEN c←a+b
         ...);
```

b)
```
IF ¬(opcode(p2)=add)
   THEN (ir←imem[pc],
         p1←ir);
         p2←p1;
         (pc←pc+1,
         IF opcode(p2)=add
           THEN c←a+b
         ...);
```

c)
```
IF ¬(opcode(ir)=add)
   THEN (ir←imem[pc],
         p1←ir);
         p2←p1;
         (pc←pc+1,
         IF opcode(p2)=add
           THEN c←a+b
         ...);
```

Abb. 5.19: Berücksichtigung von Datenabhängigkeiten

Durch die Zuweisung `p1←ir` *wird der Inhalt des Registers* `ir` *in das Register* `p1` *gespeichert. Die nachfolgende Zuweisung* `p2←p1` *kopiert den Inhalt von* `p1` *nach* `p2`*. Befindet sich zu Beginn ein* `add`*-Befehl in dem Register* `ir`*, dann ist der* `add`*-Befehl nach zwei Zeitschritten auch in* `p2` *zu finden. Der anfängliche Wert des Registers* `p2` *ist unbedeutend, da im zweiten Zeitschritt der Wert* `p2` *überschrieben wird.*

In Abbildung 5.19(c) werden die Datenabhängigkeiten berücksichtigt, so daß das Prädikat ¬(opcode(ir)=add) *an die Stelle von* $C_{kein\,Konflikt}$ *tritt. Diese Bedingung verhindert, daß die Zuweisung* c←a+b *parallel zu* pc←pc+1 *ausgeführt wird.*

5.4 Experimentelle Ergebnisse

Das Verfahren zur fall-basierten Einplanung wurde auf mehrere Prozessorarchitekturen angewendet [Hin98a, HER99, HRE99]. Tabelle 5.1 faßt die Ergebnisse, ermittelt auf einer SUN Ultra2 300 MHz, zusammen.

Architektur	Stufen der Pipeline	Anzahl Befehlsklassen	Zeit für die Synthese	Zeit für die Verifikation
3STAGE	3	1	2 sec	1 sec
ALPHA	3	10	1 min 7 sec	8 sec
DLX	5	5	15 min 32 sec	46 min 23 sec
PIC	2	17	13 min 25 sec	10 min 5 sec

Tab. 5.1: Experimentelle Ergebnisse

3STAGE ist eine dreistufige Pipeline, die an eine in [Cyr96] verifizierte einstufige Pipeline angelehnt ist. Bei dem DLX-Prozessor handelt es sich um die in [HP96] dargestellte Integer-Pipeline. Die Beschreibung des PIC basiert auf dem Mikrocontroller PIC16C5X von Microchip [Mic93] und die des ALPHA auf der Alpha-Architektur von Digital [Dig92]. In den folgenden Abschnitten werden die Syntheseergebnisse detailliert dargestellt; die Verifikation wird in Abschnitt 5.5 beschrieben. Auf die ausführliche Darstellung des ALPHA-Prozessors wird im folgenden verzichtet.

Das *TUD Transformationswerkzeug (TUDT)* plant eine dynamische Pipeline ein. Ausgehend von einer algorithmischen Beschreibung eines sequentiellen Prozessors wird eine Verhaltensbeschreibung des Pipelinesystems abgeleitet. Eine strukturelle Repräsentation kann bisher noch nicht transformativ gewonnen werden, da es z.B. Schwierigkeiten bereitet, die Operationen auf die Funktionseinheiten abzubilden. Wie werden die Eingänge verschaltet? Welche Multiplexer werden benötigt? Die Güte der bisher bestehenden Werkzeuge, die eine Verhaltenssynthese durchführen, wie z.B. der *Synopsys Behavioral Compiler* [Syn98b], zeigt die Komplexität dieses Schrittes.

Eine strukturelle Darstellung wird, wie in Abschnitt 2.6 dargestellt, mit Hilfe des *Synopsys Design Compilers* [Syn98a] gewonnen. In einer Vorverarbeitung wird die Verhaltensbeschreibung der Pipeline nach VHDL übersetzt. Dabei werden die als Arrays modellierten Speicher strukturell beschrieben (siehe Abschnitt 3.5), so daß der *Synopsys Design Compiler* eine Implementierung generieren kann.

5.4.1 Beispiel einer dreistufigen Pipeline

Aus dem sequentiellen Prozessor der Abbildung 5.21(a) wird durch Anwenden der fall-basierten Einplanung eine dreistufige Pipeline erzeugt, deren Ablaufdiagramm Abbildung 5.20 und deren Beschreibung 5.21(c) zu entnehmen ist.

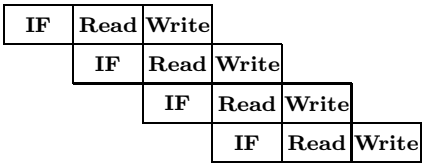


Abb. 5.20: Ablaufdiagramm der dreistufigen Pipeline

Der Prozessor führt einen Befehl aus, der sich in drei Phasen gliedert. In der ersten Phase (**I**nstruction **F**etch) wird durch `ir←mem[pc],pc←inc(pc)` ein Befehl geladen, in der zweiten (**Read**) ein Wert aus einem Register der Registerbank durch `wbreg←rf[op2(ir)]` gelesen und in der dritten (**Write**) der gelesene Wert durch `rf[op1(ir)]←wbreg` wieder abgespeichert.
In [Cyr96] wird eine einstufige Pipeline verifiziert. In dieser Pipeline führt der Pipelinezustand zwei parallele Anweisungen aus. Der sequentielle Prozessor aus Abbildung 5.21(a) führt diese beiden Anweisungen hintereinander aus, erweitert um eine Befehlsphase, in der die Befehle geladen werden.
Nach der ersten Iteration ergibt sich die in Abbildung 5.21(b) dargestellte Beschreibung. Durch die Transformation R1 wird ein neues Segment L^1 erzeugt. Im else-Zweig der if-then-else-Anweisung von L^1 wird parallel zur zweiten Befehlsphase ein neuer Befehl geladen. Ein Pipeline-Register `p0` ist erforderlich, um den Wert des Registers `ir` zwischenzuspeichern. Der then-Zweig beschreibt das Leeren der Pipeline. In der Pipeline befindet sich ein Befehl, der terminiert wird.
Durch die nächste Iteration der fall-basierten Einplanung ergibt sich die Beschreibung aus 5.21(c). Das neue Segment L^2 beschreibt den Pipelinezustand. Der else-Zweig der if-then-else-Anweisung führt alle drei Befehlsphasen parallel aus. Das Leeren der Pipeline wird durch then-Zweig beschrieben. Da sich zwei Befehle in der Pipeline befinden, realisiert dieser Zweig die letzte Befehlsphase der als vorletztes geladenen sowie die zweite und dritte Phase der zuletzt geladenen Instruktion.
Wird die zweite Phase parallel zu der dritten des vorangehenden Befehls ausgeführt, besteht eine Datenabhängigkeit, falls `p0[0]` und `op1(p0)=op2(ir)` erfüllt sind. Ein Befehl lädt den Wert aus dem Register der Registerbank, in das die vorausgehende Instruktion einen Wert abspeichert. Als Ziel dient das Register `wbreg`. Da die Befehle parallel in der Pipeline ausgeführt werden, hat der vorangehende Befehl noch nicht den aktuellen Wert aus `wbreg` in `rf` geschrieben. Der Konflikt wird gelöst, indem der Speicherzugriff verhindert wird.
Die Beschreibung 5.21(c) wird durch Anwenden von 18 Transformationen au-

a) $\mathcal{B} = (\{L^0\}, L^0, \{ir, pc, wbreg, mem, rf\})$ mit

```
L^0 :
  IF flush
    THEN stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc));
         IF ir[0]
           THEN wbreg←rf[op2(ir)];
                rf[op1(ir)]←wbreg;
         L^0;
```

b) $\mathcal{B}' = (\{L^0, L^1\}, L^0, \{ir, p0, pc, wbreg, mem, rf\})$ mit

```
L^0 :
  IF flush
    THEN stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc));
         L^1;
L^1 :
  IF flush THEN
    THEN IF ir[0]
           THEN wbreg←rf[op2(ir)];
         IF ir[0]
           THEN rf[op1(ir)]←wbreg;
         stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc),
         IF ir[0]
           THEN wbreg←rf[op2(ir)];
         p0←ir);
         IF p0[0]
           THEN rf[op1(p0)]←wbreg;
         L^1;
```

c) $\mathcal{B}'' = (\{L^0, L^1, L^2\}, L^0, \{ir, p0, pc, wbreg, mem, rf\})$ mit

```
L^0 :
  IF flush
    THEN stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc));
         L^1;
L^1 :
  IF flush THEN
    THEN IF ir[0]
           THEN wbreg←rf[op2(ir)];
         IF ir[0]
           THEN rf[op1(ir)]←wbreg;
         stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc),
         IF ir[0]
           THEN wbreg←rf[op2(ir)];
         p0←ir);
         L^2;
L^2 :
  IF flush
    THEN (IF ir[0]∧¬(p0[0]∧
                (op1(p0)=op2(ir)))
           THEN wbreg←rf[op2(ir)],
         IF p0[0]
           THEN rf[op1(p0)]←wbreg);
         IF ir[0]
           THEN rf[op1(ir)]←wbreg;
         stall;
         L^Ende;
    ELSE (ir←mem[pc],
         pc←inc(pc),
         IF ir[0]∧¬(p0[0]∧
                (op1(p0)=op2(ir)))
           THEN wbreg←rf[op2(ir)];
         p0←ir,
         IF p0[0]
           THEN rf[op1(p0)]←wbreg);
         L^2;
```

Abb. 5.21: Erzeugen einer dreistufigen Pipeline

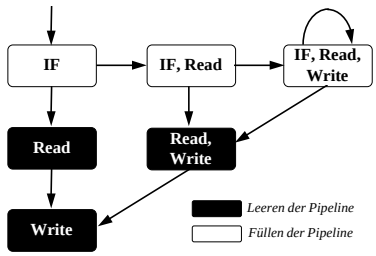


Abb. 5.22: Zustandsautomat der dreistufigen Pipeline

tomatisch erzeugt. Das Ergebnis wird in 2 Sekunden (300 MHz, Sun Ultra2) berechnet. Vorgaben von Ressourcen müssen nicht beachtet werden, so daß Algorithmus 5.1 zur fall-basierten Einplanung aus Abschnitt 5.2.3 ohne Erweiterung angewendet werden kann. Die Korrektheit der Pipeline hinsichtlich der sequentiellen Spezifikation wird in Abschnitt 5.5.2 nachgewiesen.
In Abbildung 5.22 wird der Zustandsautomat dargestellt. In den dunkel unterlegten Zuständen wird die Pipeline geleert, in den restlichen gefüllt. Ein Übergang zwischen diesen Zuständen erfolgt, falls `flush` eine `1` annimmt. Ein Zustand führt alle drei Befehlsphasen parallel aus und ist sich selbst Folgezustand. Dieser Zustand wird Pipelinezustand genannt.

5.4.2 Der DLX-Prozessor

5.4.2.1 Ein DLX-Prozessor ohne Pipelining

Der DLX-Prozessor, der von Hennessy und Patterson [HP96] entworfen worden ist, dient als Beispiel für das Erzeugen von Prozessoren mit Pipelining. Diese Architektur wurde gewählt, da sie einerseits die wesentlichen Merkmale einer RISC-Architektur wie z.B. MIPS 2000, MIPS R3000, Intel i860 und Motorola M88000 aufweist und andererseits mehrere Implementierungen der Pipeline verifiziert wurden (siehe beispielsweise [BD94, Cyr96, BM97]). Der Korrektheitsbeweis einer superskalaren DLX ist in [Bur96] zu finden.

Register	Verwendung
a	Speichern des ersten Operanden
b	Speichern des zweiten Operanden
ar	Speichern der Adresse für Speicherzugriffe
din	Laden von Daten aus dem Datenspeicher
dout	Speichern von Daten in den Datenspeicher
ir	das Befehlsregister
pc	der Befehlszähler
temp	Speichern des Ergebnisses von ALU-Operationen

Tab. 5.2: Register der DLX

Zur Vereinfachung der Darstellung werden die auszuführenden Instruktionen zu Befehlsklassen zusammengefaßt. Der Prozessor ohne Pipelining führt fünf Befehle aus:

- einen Ladebefehl `load`,
- einen Speicherbefehl `store`,
- einen ALU-Befehl `alu`,

- einen unbedingten Sprungbefehl `jump` und
- einen bedingten Verzweigebefehl `branch`.

Die Instruktionen untergliedern sich in bis zu fünf Phasen:

- **I**nstruction **F**etch, um den Befehl zu laden,
- **I**nstruction **D**ecode, um die Operanden zu laden,
- **EX**ecution, um die Operation auszuführen,
- **MEM**ory access, um Daten aus dem Speicher zu laden oder in den Speicher zu schreiben, und
- **W**rite **B**ack, um das Ergebnis in ein Register der Registerbank zu speichern.

Tabelle 5.3 gibt einen Überblick über die Befehle.

Befehle	IF	ID	EX	MEM	WB
`load`	ir←imem[pc] pc←pc+1	a←rf[ir_1] b←rf[ir_2]	ar←a+ir_k	din←dmem[ar]	rf[ir_3]←din
`store`	ir←imem[pc] pc←pc+1	a←rf[ir_1] b←rf[ir_2]	ar←a+ir_k dout←b	dmem[ar]←dout	
`alu`	ir←imem[pc] pc←pc+1	a←rf[ir_1] b←rf[ir_2]	temp←a$\circ$b		rf[ir_3]←temp
`jump`	ir←imem[pc] pc←pc+1	pc←ir_k			
`branch`	ir←imem[pc] pc←pc+1	pc←pc+ir_k für rf[ir_1]=0_{32}			

mit $\mathtt{ir}_k$ *als Konstante, mit* $\mathtt{ir}_i$ *als den i-ten Operanden mit* $1 \leq i \leq 3$ *und* 0_{32} *als binärer Vektor der Länge 32 mit dem Wert 0*

Tab. 5.3: Überblick über die Befehle der DLX

Wie in Abschnitt 5.3.1 dargestellt, kann spezifiziert werden, welche Ressourcen zur Verfügung stehen. Für den DLX-Prozessor bestehen die folgenden Vorgaben:

- die Latenzzeit soll eins betragen,
- nur eine ALU und ein Addierer sind verfügbar,
- Daten werden aus dem Datenspeicher `dmem` nur in `din` eingelesen, beschrieben wird `dmem` von `dout` und adressiert von `ar`,

- `dmem` kann gleichzeitig nur einmal gelesen und einmal beschrieben werden,
- der Befehlsspeicher `imem` kann von `ir` nur gelesen, jedoch nicht beschrieben werden,
- die Register der Registerbank `rf` werden von den Registern `a` und `b` gelesen, von `din` und `temp` beschrieben und von `ir` adressiert,
- die Register der Registerbank `rf` können bis zu zweimal parallel gelesen und einmal beschrieben werden,
- die Adressierungsart *memory indirect* ist nicht erlaubt,
- das Register `pc` dient als Befehlszähler und
- die Register `a` und `b` laden die Operanden der auszuführenden Operationen.

Die LLS-Beschreibung der DLX ist Abbildung 5.4 zu entnehmen.

5.4.2.2 Vermeiden von Leerschritten

Datenabhängigkeiten zwischen Befehlen können zu Konflikten führen. Wenn ein Befehl das Ergebnis einer vorangehenden Instruktion verarbeitet, dieses Ergebnis aber noch nicht ermittelt worden ist, muß die Bearbeitung solange ausgesetzt werden, bis das Ergebnis vorliegt. Wurde das Ergebnis zwar berechnet, aber noch nicht in ein Register der Registerbank `rf` abgespeichert, kann der Wert durch Forwarding übernommen werden. In Abschnitt 5.4.2.3.2 werden die Datenkonflikte und in 5.4.2.3.3 die Kontrollkonflikte, die in der DLX-Pipeline auftreten, und deren Lösung dargestellt.

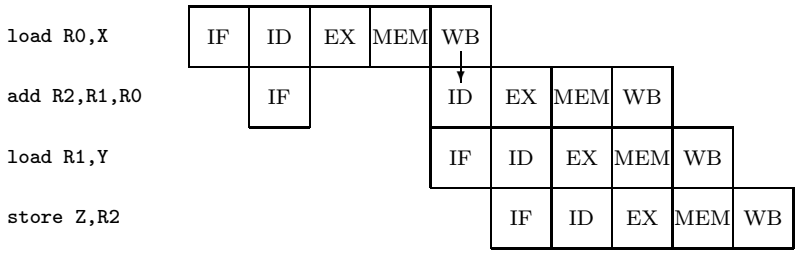


Abb. 5.23: Ein `load`-Befehl verursacht zwei Leerschritte

Wie in Abbildung 5.23 zu sehen ist, sind zwei Leerschritte erforderlich, falls ein Operand eines Befehls durch eine unmittelbar vorangehende `load`-Instruktion geladen wird. Der Pfeil deutet an, welche Befehlsphasen durch Forwarding Werte austauschen. Die DLX lädt Operanden in der ID-Phase, übernommen können die Wert aber erst in der WB-Phase des `load`-Befehls werden.
Die in der ID-Phase geladenen Operanden werden erst in der EX-Phase benötigt,

da erst in dieser eine Operation ausgeführt wird. Daher genügt es, erst in der EX-Phase den Operanden durch Forwarding zu bestimmen, so daß nur ein Leerschritt ausgeführt werden muß. Abbildung 5.24 stellt das zugehörige Ablaufdiagramm dar.

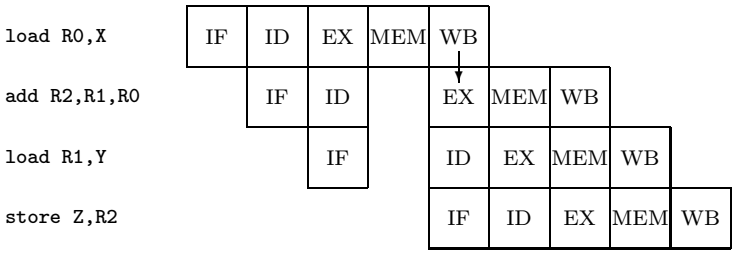


Abb. 5.24: Anhalten der Pipeline für einen Schritt

Da erst in der WB-Phase der `load`-Instruktion der geladene Wert übernommen werden kann, lädt der nachfolgende Befehl bei Adressengleichheit einen falschen Wert in die Register `a` oder `b`. Der geladene, veraltete Wert bleibt jedoch unbeachtet, weil der Operand bei Adressengleichheit durch Forwarding in der EX-Phase bestimmt wird.
Das Transformationswerkzeug erkennt die Datenabhängigkeiten zwischen der WB-Phase des `load`-Befehls und der ID-Phase des nachfolgenden Befehls und löst den Konflikt durch Einfügen zweier Leerschritte und Forwarding. Durch eine Erweiterung des Algorithmus zur Einplanung unter Ressourcenbeschränkungen kann die Anzahl der Leerschritte reduziert werden. Zu verändern ist jener Schritt, in dem die IF-, ID- und EX-Phase parallelisiert werden.

$$L^x{:}\mathrm{EX}_i;\mathrm{MEM}_i;\mathrm{WB}_i;(\mathrm{IF}_{i+2},\mathrm{ID}_{i+1});L^x \text{ wird zu } L^x{:}(\mathrm{IF}_{i+2},\mathrm{ID}'_{i+1},\mathrm{EX}_i);\mathrm{MEM}_i;\mathrm{WB}_i;L^x \text{ mit}$$

$$\mathrm{ID}_{i+1}\texttt{:}\ \begin{matrix}\texttt{a}\leftarrow\texttt{rf[ir}_{o1}\texttt{]},\\ \texttt{b}\leftarrow\texttt{rf[ir}_{o2}\texttt{]}\end{matrix} \quad \text{und } \mathrm{ID}'_{i+1}\texttt{:}\ \texttt{a}\leftarrow \begin{cases}\texttt{rf[ir}_{o1}\texttt{]}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\\ \texttt{a}\circ\texttt{b}\end{cases}, \texttt{b}\leftarrow \begin{cases}\texttt{rf[ir}_{o2}\texttt{]}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\\ \texttt{a}\circ\texttt{b}\end{cases}$$

Abb. 5.25: Parallelisieren der IF-, ID- und EX-Befehlsphase

In Abbildung 5.25 wird das Parallelisieren der drei Befehlsphasen dargestellt. Für den i-ten Befehl sind die Befehlsphasen EX_i, MEM_i und WB_i auszuführen. Es schließen sich die ID-Phase ID_{i+1} des nächsten und die IF-Phase IF_{i+2} des übernächsten Befehls daran an. Durch das Parallelisieren der IF-, ID- und EX-Phase wird es notwendig, die Zuweisungen der Register `a` und `b` zu verändern. $\texttt{p}_d$ und $\texttt{p}_t$ bezeichnen die Pipeline-Register für `din` und `temp`.
Eine neue Befehlsphase ID''_{i+1} wird derart eingefügt, daß die unveränderte ID-Phase ID_{i+1} parallel zu IF_{i+2} und EX_i ausgeführt werden kann. Aus der Folge $(\mathrm{IF}_{i+2},\mathrm{ID}'_{i+1},\mathrm{EX}_i);\mathrm{MEM}_i;\mathrm{WB}_i;$ wird $(\mathrm{IF}_{i+2},\mathrm{ID}_{i+1},\mathrm{EX}_i);\mathrm{ID}''_{i+1};\mathrm{MEM}_i;\mathrm{WB}_i;$. Da

ID''_{i+1} und MEM_i bzw. WB_i von einander unabhängig sind, kann die neue Befehlsphase an das Ende der Folge verschoben werden. Abbildung 5.26 stellt das Ergebnis der Umformungen dar.

L^x:EX_i;MEM_i;WB_i;(IF_{i+2},ID_{i+1});L^x wird zu L^x:(IF_{i+2},ID_{i+1},EX_i);MEM_i;WB_i;ID''_{i+1};L^x mit

$$\mathrm{ID}_{i+1}\text{: } \begin{matrix}\texttt{a}\leftarrow\texttt{rf[ir}_{o1}\texttt{]},\\ \texttt{b}\leftarrow\texttt{rf[ir}_{o2}\texttt{]}\end{matrix} \text{ und } \mathrm{ID}''_{i+1}\text{: } \texttt{a}\leftarrow \begin{Bmatrix}\texttt{a}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\\ \texttt{a}\circ\texttt{b}\end{Bmatrix}, \texttt{b}\leftarrow \begin{Bmatrix}\texttt{b}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\\ \texttt{a}\circ\texttt{b}\end{Bmatrix}$$

Abb. 5.26: Einführen einer neuen Befehlsphase ID''_{i+1}

In der nächsten Iteration der fall-basierten Einplanung wird ID''_{i+1} mit IF_{i+3}, ID_{i+2} und EX_{i+1} parallelisiert. ID''_{i+1} ist redundant und entfällt unter der Anwendung der Transformation C3, da sowohl ID''_{i+1} als auch ID_{i+2} die Register `a` und `b` als Ziel verwenden. EX_{i+1} und ID''_{i+1} weisen Datenabhängigkeiten auf, so daß EX_{i+1} unter Verwendung der Transformation C2 zu EX'_{i+1} wird. Durch die Umformungen werden die aktuellen Operanden bei Adressengleichheit durch Forwarding in der EX-Phase übernommen.

L^{x+1}:MEM_i;WB_i;ID''_{i+1};(IF_{i+3},ID_{i+2},EX_{i+1});L^{x+1} wird zu L^{x+1}:MEM_i;WB_i;(IF_{i+3},ID_{i+2},EX'_{i+1});L^{x+1}

und schließlich zu L^{x+1}:(IF_{i+3},ID_{i+2},EX'_{i+1},MEM_i);WB_i;L^{x+1} mit

$$\mathrm{EX}\text{: } \begin{matrix}\texttt{ar}\leftarrow\texttt{a+ir}_k,\\ \texttt{dout}\leftarrow\texttt{b},\\ \texttt{temp}\leftarrow\texttt{a}\circ\texttt{b}\end{matrix} \text{ und } \mathrm{EX}'\text{: } \texttt{ar}\leftarrow \begin{Bmatrix}\texttt{a}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\end{Bmatrix}+\texttt{ir}_k, \texttt{dout}\leftarrow \begin{Bmatrix}\texttt{b}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\end{Bmatrix}, \texttt{temp}\leftarrow \begin{Bmatrix}\texttt{a}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\end{Bmatrix}\circ\begin{Bmatrix}\texttt{b}\\ \texttt{din}\\ \texttt{p}_d\\ \texttt{temp}\\ \texttt{p}_t\end{Bmatrix}$$

Abb. 5.27: Verändern der EX-Phase

Im Anschluß wird die Pipeline entsprechend des Algorithmus zur fall-basierten Einplanung gefüllt, indem die IF-, ID-, EX- und die MEM-Phase parallelisiert werden. Das Ergebnis ist Abbildung 5.27 zu entnehmen.

5.4.2.3 Konflikte in der Pipeline

5.4.2.3.1 Strukturelle Konflikte Ein struktureller Konflikt tritt auf, wenn verschiedene Befehlsphasen auf die gleiche Ressource zugreifen. Tabelle 5.4 gibt einen Überblick über alle möglichen strukturellen Konflikte gibt, die durch Zugriffe auf die Speicher oder die ALU auftreten. Sie lassen sich lösen, indem entweder zusätzliche Funktionseinheiten zur Verfügung gestellt werden oder beim Entwurf des Befehlssatzes vermieden wird, daß in verschiedenen Befehlsphasen auf gleiche Ressourcen zugegriffen wird.

Befehls-phase	Ressource	Verwendung
IF	Speicher	Laden eines Befehls
	ALU	Inkrementieren des Befehlszählers `pc`
ID	Registerbank	Laden von bis zu zwei Operanden
EX	ALU	Ausführen der Instruktion
MEM	Speicher	Laden und Speichern von Daten
WB	Registerbank	Speichern der ALU-Ergebnisse
	Registerbank	Speichern von Daten

Tab. 5.4: Überblick über die Ressourcen der DLX

Beispiel 5.17 *Einerseits dient die ALU dem DLX-Prozessor, um den Befehlszähler* `pc` *in der IF-Phase zu inkrementieren, andererseits führt sie die Operationen der EX-Phase aus. In der Pipeline werden die IF- und die EX-Phase parallel realisiert, so daß eine ALU nicht genügt, sondern ein zusätzlicher Addierer erforderlich wird. Der Konflikt wird durch zusätzliche Hardware gelöst.*

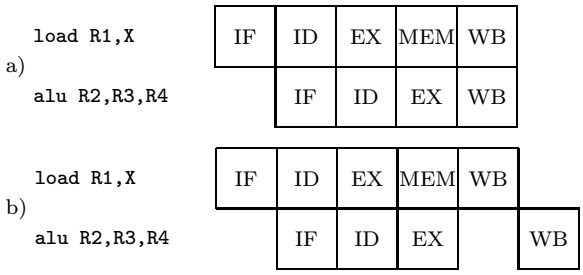


Abb. 5.28: Lösung eines Konflikts durch Verschieben einer Phase

Zwischen dem `load`*- und dem* `alu`*-Befehl kommt es zu einem weiteren strukturellen Konflikt. Die* `load`*-Instruktion besitzt fünf Phasen und schreibt in der fünften Phase den geladenen Wert in ein Register der Registerbank* `rf`*. Gleichzeitig greift der* `alu`*-Befehl auf* `rf` *zu, da die* `alu`*-Instruktion in lediglich vier Schritten ausgeführt werden kann. Abbildung 5.28(a) stellt das zugehörige Ablaufdiagramm dar. Vermieden kann der Konflikt werden, indem der* `alu`*-Befehl einen Leerschritt als MEM-Phase ausführt. Der Befehl umfaßt somit fünf Schritte, so daß kein struktureller Konflikt entsteht, wie in 5.28(b) zu sehen ist (siehe Abschnitt 5.2.2).*

Der Benutzer kann spezifizieren, welche Funktionseinheiten zur Verfügung stehen. Soll ein Konflikt durch das Bereitstellen weiterer Komponenten vermieden werden, muß der Anwender seine Vorgaben verändern. Die fall-basierte Einplanung löst strukturelle Konflikte, indem die Pipeline angehalten wird.
Das Auftreten bestimmter struktureller Konflikte kann aber bereits bei der Ermittlung der phasenorientierten Darstellung des Befehlssatzes vermieden werden (siehe Abschnitt 5.2.2). Bei der Umwandlung der befehls- in die phasenorientierte Darstellung wird eine Einplanung durchgeführt, die den Ressourcenbedarf

Ressource	Konflikt	Lösung
ALU	• ID: Inkrementieren von `pc` • EX: ALU-Operation	ein zusätzlicher Addierer
Speicher	• IF: Laden eines Befehls • MEM: Lese- oder Schreibzugriff	getrennter Daten- und Befehlsspeicher
Registerbank	• WB: `alu`-Befehl • WB: `load`-Befehl	Einfügen eines Leerschrittes als MEM-Phase des `alu`-Befehls
	• ID: alle Befehle • WB: `load`-Befehl	Möglichkeit des parallelen Lesens und Schreibens auf `rf`
	• ID: Laden zweier Operanden	Möglichkeit, zwei Lesezugriffe auf `rf` gleichzeitig auszuführen

Tab. 5.5: Überblick über die strukturellen Konflikte und ihre Lösung

reduziert (siehe Beispiel 5.17). Tabelle 5.5 nennt alle möglichen Konflikte und deren Lösung. Welche Ressourcen des DLX-Prozessors in welcher Phase benutzt werden, ist Tabelle 5.6 zu entnehmen.

Befehlsphase	Ressource	Verwendung
IF	Befehlsspeicher	Laden der Befehle
	Addierer	Inkrementieren von `pc`
ID	Registerbank	Laden von bis zu zwei Operanden
EX	ALU	Ausführen der Operationen
MEM	Datenspeicher	Laden oder Speichern von Daten
WB	Registerbank	Speichern von Daten

Tab. 5.6: Überblick über die Verwendung der Ressourcen

5.4.2.3.2 Datenkonflikte In der DLX-Pipeline werden bis zu fünf Befehle parallel ausgeführt. Greifen aufeinander folgende Befehle auf die gleichen Ressourcen oder Daten zu, kann es zu den folgenden Konflikten kommen:

- **R**ead **a**fter **W**rite
  Ein Befehl verwendet das Ergebnis einer vorangehenden Instruktion. Ein Konflikt tritt auf, wenn auf das Ergebnis zugegriffen wird, bevor es berechnet worden ist. Dieser Konflikt ist durch *Forwarding* bzw. *Bypassing* oder durch das *Anhalten der Pipeline* zu lösen.

- **W**rite **a**fter **W**rite
  Ein Wert, der von einem Befehl abgespeichert worden ist, wird von einer vorangehenden Instruktion überschrieben. Ein veralteter Wert ersetzt einen aktuell gültigen, so daß die Berechnung fehlerhaft wird.

Die Register und die Speicher des DLX-Prozessors werden jeweils nur in einer bestimmten Phase beschrieben, so daß ein WAW-Konflikt nicht auftreten kann. In der MEM-Phase wird der Datenspeicher und in der WB-Phase werden die Register der Registerbank beschrieben.

- **W**rite **a**fter **R**ead
  Eine Instruktion überschreibt einen Wert, auf den ein vorangehender Befehl noch zugreifen muß. Da der alte Wert nicht mehr zur Verfügung steht, verwendet der vorangehende Befehl einen falschen Wert.
  Bevor Daten geschrieben werden, lädt der DLX-Prozessor Werte aus den Speichern. WAR-Konflikte werden somit nicht durch Zugriffe auf die Speicher verursacht. Die Register laden ihre Werte, bevor diese verwendet werden. Wird der Wert eines Registers in mehreren Phasen gelesen, tritt ein WAR-Konflikt auf. So wird z.B. das Befehlsregister `ir` in der IF-Phase beschrieben und in allen übrigen Befehlsphasen gelesen. Ein Konflikt tritt bereits durch das Laden des nächsten Befehls auf. Durch *Einführen von Pipeline-Registern* läßt sich der Konflikt lösen.

Einführen von Pipeline-Registern Ein Pipeline-Register wird eingeführt, wenn ein Register in mehr als einer Befehlsphase gelesen wird. Bevor der Wert des Registers überschrieben wird, speichert ein Pipeline-Register den Wert, so daß er auch in nachfolgenden Befehlsphasen zur Verfügung steht.

Register	Befehlsphasen				
`a`	-	-	EX	-	-
`ar`	-	-	-	MEM	-
`b`	-	-	EX	-	-
`din`	-	-	-	-	WB
`dout`	-	-		MEM	-
`ir`	-	ID	EX	MEM	WB
`pc`	IF	ID	EX	MEM	WB
`temp`	-	-	-	MEM	WB

Tab. 5.7: Lebenszeit der verwendeten Register

Tabelle 5.7 gibt die Lebenszeit für die Register des DLX-Prozessors ohne Pipelining an. Ein Register wird einen Schritt, nachdem ein Wert abgespeichert worden ist, als "lebendig" bezeichnet. Das Register bleibt bis einschließlich zu dem Schritt lebendig, in dem der Wert zum letzten Mal verwendet wird.
Ist ein Register in mehreren Befehlsphasen lebendig, wird ein Pipeline-Register benötigt. Das Register `pc` stellt eine Ausnahme dar. Es ist in allen Phasen lebendig, da der von `pc` gespeicherte Wert benutzt wird, um die nächste Instruktion zu adressieren. Der Befehlszähler `pc` bleibt im folgenden unbeachtet (siehe Kontrollkonflikte Abschnitt 5.4.2.3.3).

Wie Tabelle 5.8 zu entnehmen ist, werden zusätzliche Register für `ir` und `temp` notwendig. Schließt sich an zwei aufeinanderfolgende `load`-Befehle ein `alu`-Befehl an, der beide geladenen Werte verwendet, wird ein weiteres Pipeline-Register für `din` erforderlich (siehe Abschnitt 5.5.5).

Register	Befehlsphasen			
	ID	EX	MEM	WB
`ir`	p_{i1} ←`ir` ...←..`ir`..	p_{i2} ←p_{i1} ...←..p_{i1}..	p_{i3} ←p_{i2} ...←..p_{i2}..	...←..p_{i3}..
`temp`			p_t ←`temp` ...←..`temp`..	...←..p_t..
`din`		...←..p_d..		p_d ←`din`

Tab. 5.8: Überblick über die eingeführten Pipeline-Register

Erforderliche Pipeline-Register werden automatisch während des Transformationsprozesses eingeführt. Ohne diese zusätzlichen Register ist das Parallelisieren bestimmter Befehlsphasen nicht möglich, da gegen die Bernsteinschen Bedingungen [Ber66] verstoßen würde. Um die Datenabhängigkeiten zwischen den Befehlsphasen durch das Einführen von Pipeline-Registern zu lösen, wird Transformation C1 angewendet.

Forwarding RAW-Konflikte können durch Forwarding gelöst werden. Verarbeitet ein Befehl die Ergebnisse einer vorangehenden Instruktion, deren Ausführung noch nicht terminiert ist, deren Ergebnisse aber bereits berechnet worden sind, kann durch Forwarding ein Datenkonflikt vermieden werden.

a)
```
rf[p0[6:10]]←din;
  (a←rf[ir[6:10]],
  b←rf[ir[11:15]]);
```

b)
```
(rf[p0[6:10]]←din,
IF (p0[6:10]=ir[6:10])
  THEN a←din,
  ELSE a←rf[ir[6:10]],
IF (p0[6:10]=ir[11:15])
  THEN b←din,
  ELSE b←rf[ir[11:15]]);
```

Abb. 5.29: Forwarding von Daten

Beispiel 5.18 *In Abbildung 5.29 sind die sequentiellen Anweisungen von einander abhängig, falls* `p0[6:10]=ir[6:10]` *oder* `p0[6:10]=ir[11:15]` *gültig ist. Da der Wert, der im Fall einer Adressengleichheit geladen werden muß, bereits in* `din` *verfügbar ist, können* `a←rf[ir[6:10]]` *und* `b←rf[ir[11:15]]` *parallel ausgeführt werden. Ohne das "Vorwegnehmen" bzw. Forwarding des Wertes von* `din` *können die Anweisungen nicht parallelisiert werden.*

In der ID-Phase lädt der DLX-Prozessor die Operanden der Befehle in die Register `a` und `b`. Tritt eine Datenabhängigkeit zwischen zwei Befehlen auf, werden die

Daten in der EX-Phase übernommen (siehe Abschnitt 5.4.2.2). In der ID-Phase werden somit in bestimmten Fällen nicht die korrekten Werte geladen. Da der DLX-Prozessor die Operationen erst in der EX-Phase ausführt, berücksichtigen diese die aktuellen Werte.

Beispiel 5.19 *Abbildung 5.30 gibt ein Beispiel. Der Befehl* `sub R3,R4,R0` *benutzt das durch* `add R0,R1,R2` *berechnete Ergebnis als Operand. Aber auch wird* `R0` *von dem Befehl* `xor R5,R6,R0` *referenziert. Der Wert von* `R0` *ist bereits berechnet, aber noch in keinem Register der Registerbank* `rf` *verfügbar, so daß der* `sub`- *und der* `xor`*-Befehl den Operand durch Forwarding übernehmen. Im Ablaufdiagramm wird Forwarding durch vertikale Pfeile angedeutet.*

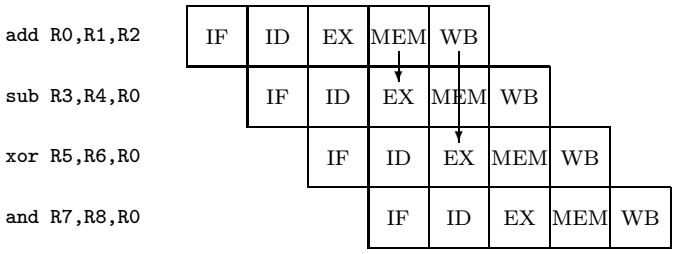


Abb. 5.30: Forwarding

Wird ein Operand eines Befehls durch eine unmittelbar vorangehende `load`-Instruktion geladen bzw. durch einen `alu`-Befehl erst berechnet, so treten RAW-Konflikte auf. Im ersten Fall steht der geladene Wert in der WB-Phase im Register `din` zur Verfügung, im zweiten ist der berechnete Wert in der EX-Phase am ALU-Ausgang oder in der MEM- bzw. WB-Phase in den Registern `temp` bzw. dessen Pipeline-Register $\mathtt{p}_t$ verfügbar. Die Tabelle 5.9 gibt einen Überblick.

Befehl	Befehlsphasen		
	EX	MEM	WB
`load`			`din`
`alu`	ALU-Ausgang	`temp`	$\mathtt{p}_t$

Tab. 5.9: Überblick über die durch Forwarding berücksichtigten Quellen

Da auf den Wert, der durch einen `load`-Befehl geladen wird, erst in der WB-Phase zugegriffen werden kann, muß, falls das Ergebnis in den vorangehenden Befehlsphasen benötigt wird, die Pipeline durch Einfügen eines Leerschrittes angehalten werden. Diese Datenabhängigkeit kann nicht durch Forwarding gelöst werden.
Werden Datenabhängigkeiten während des Parallelisierens von Befehlsphasen ermittelt, die durch Forwarding gelöst werden können, wird die Transformation C2 ausgeführt.

Anhalten der Pipeline Wenn ein Datenkonflikt nicht durch Forwarding gelöst werden kann, wird die Pipeline angehalten bzw. die Bearbeitung der nachfolgenden Befehle wird ausgesetzt, bis das Ergebnis durch Forwarding übernommen werden kann.

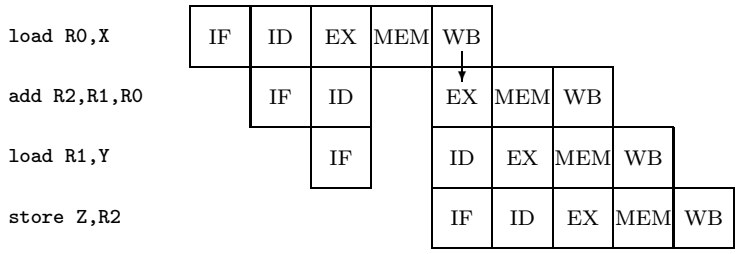


Abb. 5.31: Anhalten der Pipeline für einen Schritt

Ein Anhalten der Pipeline wird für einen Schritt erforderlich, wenn ein Operand eines `alu`-Befehls durch eine unmittelbar vorangehende `load`-Instruktion bestimmt wird. Abbildung 5.31 ist das entsprechende Ablaufdiagramm zu entnehmen. Da das Ergebnis des `load`-Befehls erst in der WB-Phase durch Forwarding übernommen werden kann, muß ein Leerschritt ausgeführt werden.

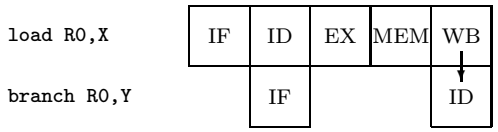


Abb. 5.32: Anhalten der Pipeline für zwei Schritte

Die Pipeline wird dagegen für zwei Leerschritte angehalten, wenn ein `branch`-Befehl die Verzweigung von einem Wert abhängig macht, der durch eine vorangehende `load`-Instruktion geladen wird. Da der `branch`-Befehl keine EX-Phase besitzt, muß der Wert bereits in der ID-Phase verfügbar sein. Wie in Abbildung 5.32 zu sehen ist, werden zwei Leerschritte erforderlich.
Werden Befehlsphasen parallelisiert, ist zu überprüfen, ob das gleichzeitige Ausführen der Befehlsphasen die durch den Benutzer vorgegebenen Einschränkungen verletzen. Findet das Transformationswerkzeug eine Verletzung, werden Leerschritte automatisch eingefügt (siehe Abschnitt 5.3.2).

5.4.2.3.3 Kontrollkonflikte Kontrollkonflikte werden in der Pipeline durch Sprung- und Verzweigebefehle verursacht. Während der `jump`-Befehl einen unbedingten Sprung ausführt, realisiert die `branch`-Instruktion eine bedingte Verzweigung. Beide Befehle verändern den Wert des Befehlszählers `pc` in der ID-Phase. Wird parallel ein neuer Befehl adressiert, besteht eine Datenabhängigkeit zwischen dem Sprung- bzw. Verzweigebefehl und der unmittelbar sich anschließenden Instruktion, wie Abbildung 5.33 zu entnehmen ist. Die Datenabhängigkeit

wird durch den Pfeil angedeutet. Es handelt sich um einen RAW-Konflikt, der durch Forwarding oder Anhalten der Pipeline gelöst werden kann.

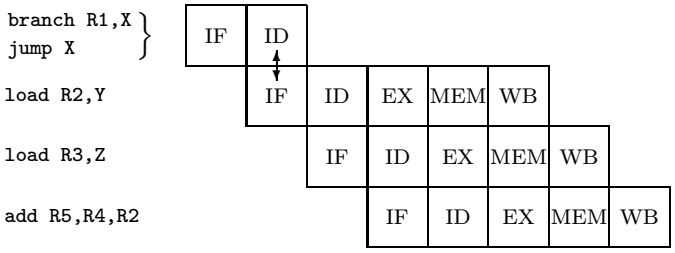


Abb. 5.33: Kontrollkonflikt

Unbedingte Sprünge Kontrollkonflikte, die durch unbedingte Sprünge verursacht werden, können durch Forwarding gelöst werden, da sowohl die Tatsache, daß ein Sprung auszuführen ist, als auch das Sprungziel bekannt sind. Das Sprungziel muß nicht berechnet werden, sondern ist als Konstante gegeben.

Register	Forwarding	Bedingung
ir	imem[ir_k]	jump-Befehl in der ID-Phase
	imem[pc]	sonst
pc	inc(ir_k)	jump-Befehl in der ID-Phase
	inc(pc)	sonst

Tab. 5.10: Forwarding zur Lösung eines Kontrollkonfliktes

Das Transformationswerkzeug unterscheidet nicht zwischen Daten- und Kontrollkonflikten. Es bestehen Datenabhängigkeiten zwischen dem unbedingten Sprungbefehl und der IF-Phase des nächsten Befehls, die durch Forwarding gelöst werden können. Die Transformation C2 löst die Abhängigkeit durch Forwarding. Eine Verzweigung wird eingeführt, so daß, falls ein jump-Befehl vorausgeht, anstelle des Befehlszählers pc die Konstante ir_k verwendet wird. Die Tabelle 5.10 gibt einen Überblick.

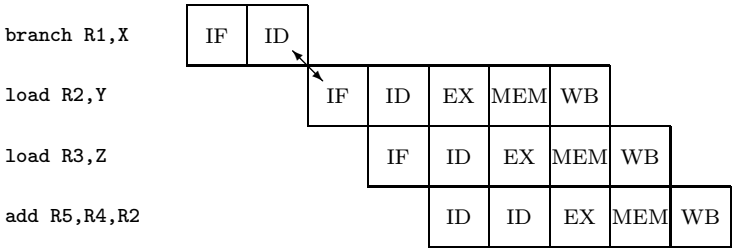


Abb. 5.34: Lösen eines Kontrollkonfliktes durch Anhalten der Pipeline

Bedingte Verzweigungen Im Unterschied zu dem unbedingten Sprung muß in der ID-Phase des `branch`-Befehls entschieden werden, ob eine Verzweigung realisiert wird. Das Sprungziel wird relativ zu dem Wert von `pc` angegeben, so daß eine Addition zur Berechnung der Zieladresse durchzuführen ist.
Der Kontrollkonflikt wird durch das Einfügen eines Leerschrittes gelöst (siehe Abschnitt 5.3.2), was in Abbildung 5.34 anhand eines Beispiels dargestellt wird. Die Pipeline wird jedoch nur angehalten, wenn die Verzweigung ausgeführt wird.

5.4.2.4 Der DLX-Prozessor mit Pipelining

Die Verhaltensbeschreibung der DLX mit Pipelining umfaßt neun verschiedene Segmente. Abbildung 5.35 stellt den Zustandsautomat dar. Jeder der Zustände repräsentiert eine bestimmte Kombination der Befehlsphasen in der Pipeline. Tabelle 5.11 gibt einen Überblick über die Zustände und die in den Zuständen ausgeführten Befehlsphasen. Zur Vereinfachung der Darstellung beschreibt der Zustandsautomat nur das Füllen der Pipeline.

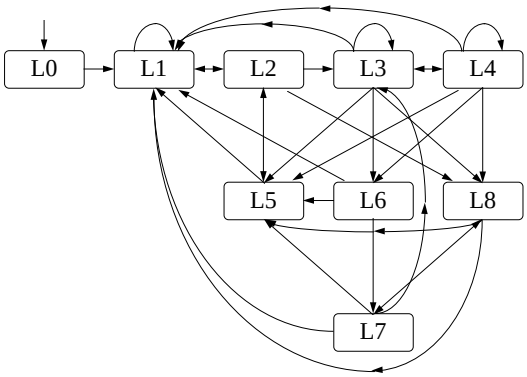


Abb. 5.35: Zustandsautomat der DLX für das Füllen der Pipeline

Das Laden des ersten Befehls wird durch das Segment L0 beschrieben, welches als Startzustand des Automaten dient. Sukzessive werden weitere Befehle durch die Segmente bzw. Zustände L1 bis L3 geladen. L4 beschreibt den Pipelinezustand, in dem fünf Befehle parallel ausgeführt werden. In den Zuständen L5 bis L8 sind vorangehende Befehle, die keine fünf Befehlsphasen besitzen, bereits terminiert, so daß bestimmte Phasen entfallen. Da der `branch`-Befehl nur aus zwei Befehlsphasen besteht, entfällt in den Segmenten L5 bis L8 die EX- bzw. MEM-Phase. In jedem Zustand wird ein Befehl geladen. Abgesehen von L0 befinden sich zeitgleich noch mehrere Befehle in der Pipeline. Die Zustände beschreiben das parallele Ausführen einer bestimmten Kombination von Befehlsphasen. Treten Konflikte in der Pipeline beim Ausführen der bereits geladenen Instruktionen auf, die das Anhalten der Pipeline erforderlich machen, beschreibt das Segment alle Schritte einschließlich des Ladens des nächsten Befehls.

Zustand	Anzahl Befehle	Befehlsphasen	Bemerkung
L0	1	IF	Füllen der Pipeline
L1	2	IF, ID	Füllen der Pipeline
L2	3	IF, ID, EX	Füllen der Pipeline
L3	4	IF, ID, EX, MEM	Füllen der Pipeline
L4	5	IF, ID, EX, MEM, WB	Pipelinezustand
L5	3	IF, ID, WB	`branch` wäre in der EX-Phase; wegen eines Leerschrittes befindet sich kein Befehl in der MEM-Phase
L6	4	IF, ID, MEM, WB	`branch` wäre in der EX-Phase
L7	4	IF, ID, EX, WB	`branch` wäre in der MEM-Phase
L8	3	IF, ID, MEM	`branch` wäre in der EX-Phase; die WB-Phase des fünften Befehls ist während des Anhaltens der Pipeline bereits ausgeführt worden

Tab. 5.11: Tabelle der Zustände der DLX

Beispiel 5.20 *Abbildung 5.36 ist zu entnehmen, welches Verhalten durch das Segment bzw. den Zustand L2 beschrieben wird, in dem bereits zwei Instruktionen geladen sind. Treten keine Konflikte auf, füllt sich die Pipeline, indem ein dritter Befehl geladen wird. Wurde zuvor ein* `branch`*-Befehl geladen, entfällt die ID-Phase, falls die Verzweigung nicht ausgeführt wird.*
Andernfalls ist es unter der Bedingung, daß keine Datenabhängigkeiten zwischen den vorausgehenden Befehlen bestehen, nötig, die Pipeline für einen Schritt anzuhalten, um die Sprungadresse zu berechnen. Aus diesem Grund werden zuerst die ID-Phase des `branch`*-Befehls und die EX-Phase der zuvor geladenen Instruktion ausgeführt. Während im nächsten Schritt der nächste Befehl geladen wird, führt die Pipeline gleichzeitig die MEM-Phase des vorangehenden Befehls aus. Der* `branch`*-Befehl ist bereits terminiert.*

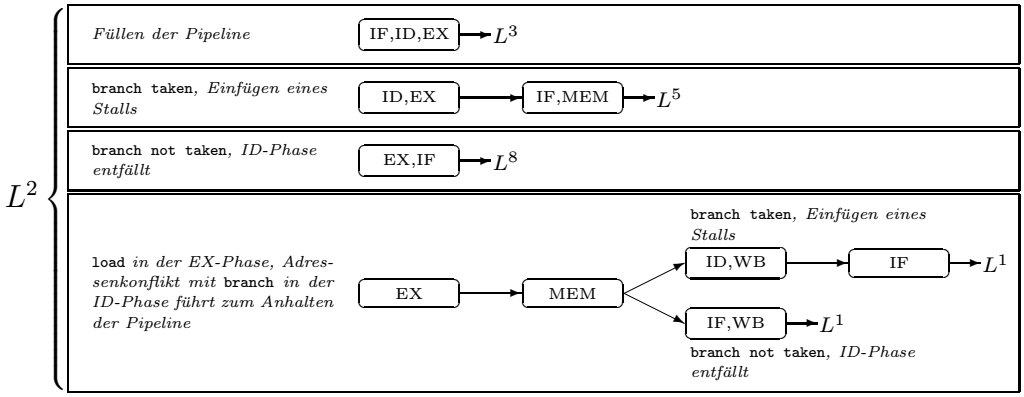


Abb. 5.36: Zustand L^2 der DLX-Pipeline

Für den Fall, daß Datenabhängigkeiten existieren, müssen zwei weitere Stalls

eingefügt werden. Lädt ein der `branch`*-Instruktion unmittelbar vorangehender* `load`*-Befehl erst den Operanden, der durch den* `branch`*-Befehl auszuwerten ist, wird die Pipeline für zwei Schritte angehalten. Nacheinander werden die EX- und die MEM-Phase des* `load`*-Befehls ausgeführt. Da der Operand in der MEM-Phase bereits in das Register* `din` *geladen worden ist, kann der* `branch`*-Befehl, falls er ausgeführt wird, parallel zu der WB-Phase realisiert werden. Erst im Anschluß daran kann der nächste Befehl geladen werden. Wird die Verzweigung nicht ausgeführt, können die WB-Phase des* `load`*-Befehls und die IF-Phase parallel abgearbeitet werden.*

Der gesamte Transformationsprozeß benötigt 21 Minuten (300 MHz, Sun Ultra2). Das Entdecken von Konflikten in der Pipeline, die mit den zur Verfügung stehenden Ressourcen nicht gelöst werden können, (siehe Abschnitt 5.3.3), das Lösen der Konflikte (siehe Abschnitt 5.3.4) und das Vereinfachen während und nach Abschluß des Transformationsprozesses (siehe Kapitel 7) sind die zeitaufwendigsten Operationen. Bis zu 1060 Transformationen sind notwendig, um die DLX mit Pipelining zu erzeugen.

5.4.2.5 CPI des DLX-Prozessors mit Pipelining

Eine Implementierung eines Prozessors läßt sich einerseits durch die Kosten, die seine Realisierung verursacht, andererseits durch die Zeit, die für das Ausführen eines Programmes benötigt wird, charakterisieren. Da die Anzahl der Funktionseinheiten für den DLX-Prozessor mit Pipelining vorgegeben worden ist, wird im folgenden eine Bewertung durch die *Clock cycles per Instructions (CPI)* [HP96] vorgenommen. Die mittlere Schrittanzahl für die Ausführung eines Befehls läßt sich entsprechend Formel 5.1 [HP96] berechnen.

$$CPI_{seq} = \sum_{alle\ Befehle} Wahrscheinlichkeit \cdot Anzahl\ der\ Schritte \qquad (5.1)$$

Die CPI eines Prozessors ist stark von dem jeweils betrachteten Programm abhängig. Die verschiedenen Befehle bzw. Befehlsklassen benötigen im allgemeinen unterschiedlich viele Schritte, um die Operationen auszuführen. Die Wahrscheinlichkeit für das Auftreten eines Befehls und somit der Beitrag am CPI, den ein Befehl erbringt, hängt jedoch von dem betrachteten Programm ab, so daß für unterschiedliche Programme die Werte stark differieren können. Im folgenden sind die Wahrscheinlichkeiten für das Auftreten der Befehle [HP96] entnommen. Tabelle 5.12 stellt die Wahrscheinlichkeiten für das Auftreten der verschiedenen Befehle des DLX-Prozessors dar. Für die sequentielle DLX berechnet sich nach Formel 5.1 eine mittlere Taktanzahl pro Befehl von 3,48. In [HP96] ergibt sich für die sequentielle DLX ein Wert von 4,44.
Formel 5.2 gibt an, wie die CPI für einen Prozessor mit Pipelining berechnet werden kann. Die ideale CPI für eine Pipeline liegt bei 1, da ohne Anhalten der

Befehlsklasse	Wahrscheinlichkeit für das Auftreten	Anzahl der Schritte	CPI_{seq}
load	20%	5	1,00
store	10%	4	0,40
alu	40%	4	1,60
jump	5%	2	0,10
branch taken	13%	2	0,26
branch untaken	12%	1	0,12
			3,48

Tab. 5.12: CPI der sequentiellen DLX

Pipeline in jedem Schritt ein neuer Befehl geladen wird. Treten Stalls auf, muß die Wahrscheinlichkeit für ihr Auftreten zu dem idealen CPI addiert werden.

$$CPI_{pipe} = CPI_{ideal} + CPI_{stall} = 1 + CPI_{stall} \tag{5.2}$$

Die Wahrscheinlichkeit für das Auftreten eines Leerschrittes hängt jedoch stark von dem betrachteten Programm ab. Werden, um den Code zu optimieren, Compiler-Techniken, wie z.B. das Entrollen von Schleifen, Software Pipelining oder Trace Scheduling [HP96] verwendet sinkt die Anzahl der Leerschritte. Aus diesem Grund wird im folgenden nur eine Abschätzung für den DLX-Prozessor mit Pipelining gegeben.

Befehlsklasse	Wahrscheinlichkeit für das Auftreten	Wahrscheinlichkeit eines Leerschrittes	Anzahl von Leerschritten	CPI_{stall}
load	20%	$\approx$ 25%	1 2	0,05 - 0,10
taken branch	13%	45% - 100%	1	0,06 - 0,13
				0,11 - 0,23

Tab. 5.13: CPI des DLX-Prozessors mit Pipelining

Entsprechend Formel 5.2 und Tabelle 5.13 ergibt sich ein CPI von:

$$1{,}11 \leq CPI_{pipe} \leq 1{,}23$$

Der DLX-Prozessor mit Pipelining nach [HP96] besitzt ein CPI von 1,11. Im Vergleich mit der automatisch generierten DLX-Pipeline zeigt sich, daß beide Prozessoren ein identisches Stall-Verhalten und somit einen identischen CPI besitzen.
Die Beschleunigung eines Prozessors durch Pipelining ergibt sich entsprechend der Formel 5.3 als Quotient aus der CPI_{seq} des Prozessors ohne und der CPI_{pipe}

des Prozessors mit Pipelining [HP96].

$$SPEEDUP = \frac{CPI_{seq}}{CPI_{pipe}} \tag{5.3}$$

Für den DLX-Prozessor mit Pipelining ergibt sich gemäß der Formel 5.3 eine Beschleunigung von:

$$2{,}83 = \frac{3{,}48}{1{,}23} \leq SPEEDUP \leq \frac{3{,}48}{1{,}11} = 3{,}14$$

In [HP96] wird ein SPEEDUP von (4,5/1,11) = 4,05 angegeben. Der höhere Wert resultiert aus einer höheren CPI_{seq}.

5.4.3 Ein PIC-Prozessor

Ein PIC-Prozessor ist ein einfacher Mikrocontroller, dessen Befehle in zwei Schritten abgearbeitet werden. Das Beispiel basiert auf dem Prozessor PIC16C5X von Microchip [Mic93]. Der Prozessor besitzt folgende Register:

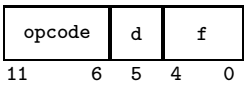
- ein Befehlsregister `ir[11:0]`,
- einen Befehlszähler `pc[11:0]`,
- ein Arbeitsregister `w[7:0]`,
- drei Ein- und Ausgaberegister `port a[3:0]`, `b[7:0]` und `c[7:0]` und
- drei Kontrollregister `trisa[3:0]`, `trisb[7:0]` und `trisc[7:0]` für die Ein- und Ausgaberegister.

und Speicher:

- einen Befehlsspeicher `rom[511:0;11:0]`,
- eine Registerbank `rf[24:0;11:0]` und
- einen Stack `stack[1:0;11:0]`.

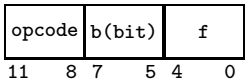
Der Maschinenbefehlssatz umfaßt 17 Instruktionen, deren Wirkungsweise Tabelle 5.15 darstellt. Um die Darstellung zu vereinfachen, werden die arithmetisch logischen Befehle zu einem Befehl *alu* zusammengefaßt. Es existieren drei Befehlsformate:

- Byte-orientierte Befehle:

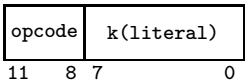
  

  Das Bit d, `des(d)` abgekürzt, entscheidet, ob das Ergebnis in das Arbeitsregister `w` (`d`=0) oder in `res(f)` (`d`=1) gespeichert werden muß. Die Bit

f spezifizieren das Register, `res(f)` abgekürzt, dessen Inhalt als Operand dient und in das, falls `d`=1 gilt, das Ergebnis abgespeichert wird. Tabelle 5.14 kann entnommen werden, welches f welches Register auswählt.

- Bit-orientierte Befehle: 

Die Bit b geben an, welches Bit des Registers f zu manipulieren ist.

- Konstanten- und Kontrollbefehle: 

Konstantenbefehle verknüpfen das Arbeitsregister `w` und Kontrollbefehle den Befehlszähler `pc` mit einer Konstanten `k`.

f	Register
00001	Time Clock/Counter Register `rtcc`
00010	Befehlszähler `pc`
00011	Statusregister `status`
00100	Speicher Auswahl Register `fsr`
00101	Ein- \ Ausgabe `port a`
00110	Ein- \ Ausgabe `port b`
00111	Ein- \ Ausgabe `port c`
01000	`rf[0]`
01001	`rf[1]`
...	...
11111	`rf[24]`

Tab. 5.14: Registerauswahl

Da für zehn Befehle jedes Register, abgesehen von dem Befehlsregister `ir`, den Kontrollregistern für die Ein- und Ausgabe `trisa` bis `trisc` und dem Arbeitsregister `w`, als Quell- und Zielregister dienen kann, ist in der LLS-Beschreibung die Registerbank `rf` um sieben Register erweitert. Der Befehlszähler `pc` wird z.B. als `rf[2]` und `port a` als `rf[5]` modelliert. Das Register `w` bleibt erhalten, da `w` nicht durch `f` kodiert, sondern durch `d=0` angesprochen wird.
Die Modellierung der Register als Erweiterung der Registerbank `rf` erschwert nicht nur die Synthese sondern auch die Verifikation des PIC-Prozessors (siehe Abschnitt 5.5.4). Aus diesem Grund wurden mehrere verschiedene Versionen des PIC-Prozessors synthetisiert. Um die Komplexität zu verringern, wurde der Befehlszähler `pc` und das Statusregister `status` unabhängig von `rf` modelliert (PIC 1). Somit konnten diese beiden Register nicht durch `f` kodiert werden. In 3 Minuten (300 MHz, Sun Ultra2) wurde unter Anwendung von 67 Transformationen eine Verhaltensbeschreibung einer zweistufigen Pipeline erzeugt.
Im nächsten Schritt wurde eine Beschreibung einer Pipeline (PIC 2), in der `status` als `rf[3]` modelliert wurde, erzeugt. Die Synthesezeit blieb annähernd

Befehl	Syntax	Operation
alu w,f	*alu*`wf f,d`	$des(d) \leftarrow res(f) \circ$ `w`, `status[2:0]` setzen;
alu literal w	*alu*`lw k`	`w` $\leftarrow$ `w` $\circ k$, `status[2]` setzen;
bit clear f	`bcf f,b`	$res(f)[b] \leftarrow$ `0B1`;
bit set f	`bsf f,b`	$res(f)[b] \leftarrow$ `1B1`;
bit test, skip if clear	`btfsc f,b`	Abbrechen des nächsten Befehls für $res(f)[b] =$ `0B1`;
bit test, skip if set	`btfss f,b`	Abbrechen des nächsten Befehls für $res(f)[b] =$ `1B1`;
call subroutine	`call k`	`stack`[0] $\leftarrow$ `pc` $+ 1$, `stack`[1] $\leftarrow$ `stack`[0], `pc` $\leftarrow$ `status`$[7:5]\#k$;
clear f	`clrf f`	$res(f) \leftarrow 0$;
clear w	`clrw w`	`w` $\leftarrow 0$, `status[2]` setzen;
decrement f, skip if 0	`decfsz f,d`	$des(d) \leftarrow res(f) - 1$, für $des(d) = 0$ Abbrechen des nächsten Befehls;
unconditional branch	`goto k`	`pc` $\leftarrow$ `status`$[7:5]\#k$;
increment f, skip if 0	`incfsz f,d`	$des(d) \leftarrow res(f) + 1$, für $des(d) = 0$ Abbrechen des nächsten Befehls;
move f	`movf f`	$des(d) \leftarrow res(f)$;
move w to f	`movwf f`	$res(f) \leftarrow$ `w`, `status[2]` setzen;
move literal to w	`movlw k`	`w` $\leftarrow k$;
return with literal in w	`retlw k`	`w` $\leftarrow k$, `pc` $\leftarrow$ `stack`[0], `stack`[0] $\leftarrow$ `stack`[1];
tris f	`tris f`	`trisa` $\leftarrow$ `w` für $f = 5$, `trisb` $\leftarrow$ `w` für $f = 6$, `trisc` $\leftarrow$ `w` für $f = 7$;

Tab. 5.15: Maschinenbefehlssatz

konstant bei 3 Minuten (300 MHz, Sun Ultra2), obwohl sechs Befehle `status` bzw. `rf[3]` ansprachen. Während die Anzahl der Segmente unverändert bleibt, erhöhte sich die Anzahl der Transformationen um eins.
Die Synthesezeit erhöhte sich dagegen auf 10 Minuten (300 MHz, Sun Ultra2), wenn `pc` als `rf[2]` modelliert (PIC 3) wird, da acht weitere Befehle den Befehlszähler veränderten und Programmsprünge verursachten. Die Anzahl der Segmente blieb konstant, die Anzahl der Transformationen erhöhte sich auf 87.
Wurden sowohl `pc` als auch `status` als Register der Registerbank modelliert (PIC 4), ergabt sich durch Anwenden von 92 Transformationen eine Laufzeit von 14 Minuten (300 MHz, Sun Ultra2). Diese Beschreibung entspricht dem Mikrocontroller von Microchip. Tabelle 5.16 faßt die Ergebnisse zusammen.

PIC	die auf `rf` abgebildeten Register		Transformationen	Segmente	Zeit für die Synthese
	Register	Anzahl			
1	`fsr`, `port a` bis `c`, `rtcc`	5	67	4	2 min 46 sec
2	`fsr`, `port a` bis `c`, `rtcc`, `status`	6	68	4	2 min 57 sec
3	`fsr`, `pc`, `port a` bis `c`, `rtcc`	6	87	4	9 min 7 sec
4	`fsr`, `pc`, `port a` bis `c`, `rtcc`, `status`	7	92	4	13 min 25 sec

Tab. 5.16: Exploration des Entwurfsraumes

Im Vergleich zu der fünfstufigen DLX-Pipeline ist die Laufzeit der Synthese des PIC-Prozessors verhältnismäßig hoch. Die Möglichkeit des PIC-Prozessors, durch `f` das Quell- und Zielregister zu kodieren, und der im Vergleich zu den vorangehenden Beispielen große Befehlssatz machen es notwendig, viele Datenabhängigkeiten zu beachten und viele Fallunterscheidungen durchzuführen. So können nicht nur die Kontrollbefehle, sondern auch zehn weitere Befehle (`f=2`) auf den Befehlszähler zugreifen. Außerdem brechen vier Befehle die Ausführung des nächsten Befehls für den Fall ab, daß ein bestimmtes Ergebnis erzielt wird. Aber nicht nur die Lösung der auftretenden Abhängigkeiten ist aufwendiger als in den vorangegangenen Beispielen, sondern auch das Erkennen dieser.
Eine verhältnismäßig hohe Laufzeit ergibt sich aber auch durch eine aufwendige Vereinfachung der Beschreibung (siehe Kapitel 7). So unterscheiden sich z.B. die Längen der Befehlscodes. Während Byte-orientierte Befehle einen Befehlscode von sechs Bit besitzen, und besteht er bei Bit-orientierten sowie Konstanten- und Kontrollbefehlen aus vier, vereinzelt auch aus drei Bit (z.B. der `goto`-Befehl). Desweiteren müssen einzelne Bit betrachtet werden, wie z.B. das Bit `d`.

5.5 Verifikation eines Prozessors mit Pipelining

5.5.1 Formale Verifikation durch symbolische Simulation

Ein Prozessor mit Pipelining läßt sich durch Methoden der Äquivalenzprüfung [BD94, Cyr96, REH99, RHE99a] formal verifizieren. Durch symbolische Simulation aller möglichen Pfade überprüft ein am Lehrstuhl entwickelter Äquivalenzprüfer [REH99, RHE99a] die Äquivalenz der sequentiellen Spezifikation und der Implementierung der Pipeline. Der Äquivalenzprüfer ist von dem *TUD Transformationswerkzeug (TUDT)* unabhängig und wird verwendet, um die vom Transformationswerkzeug generierten Ergebnisse formal zu verifizieren. Die Verifikation wird durchgeführt, um *Implementierungsfehler* des Synthesewerkzeugs aus-

findig zu machen. Im Gegensatz zur "klassischen" Simulation ist die symbolische Simulation vollständig, da ein symbolischer Pfad eine Vielzahl "klassischer" Simulationsläufe zusammenfaßt.
Das Verifikationsproblem läßt sich nach dem Verfahren von Burch und Dill [BD94] auf die Überprüfung der Äquivalenz zweier finiter azyklischer Sequenzen [EHR98, HER99, REH99, RHE99a] zurückführen. Die folgenden beiden Sequenzen sind zu vergleichen:

- Das Pipelinesystem wird geleert (*Flushing*), um anschließend den letzten Befehl sequentiell auszuführen. Im folgenden wird diese Sequenz als *Spezifikation* bezeichnet.
- Bevor die Pipeline geleert wird (*Flushing*), startet die Pipeline einen neuen Befehl. Diese Sequenz wird im folgenden *Implementierung* genannt.

Im ersten führt der sequentielle Prozessor die Berechnung aus, im zweiten Fall hingegen realisiert das Pipelinesystem den zuletzt geladenen Befehl.

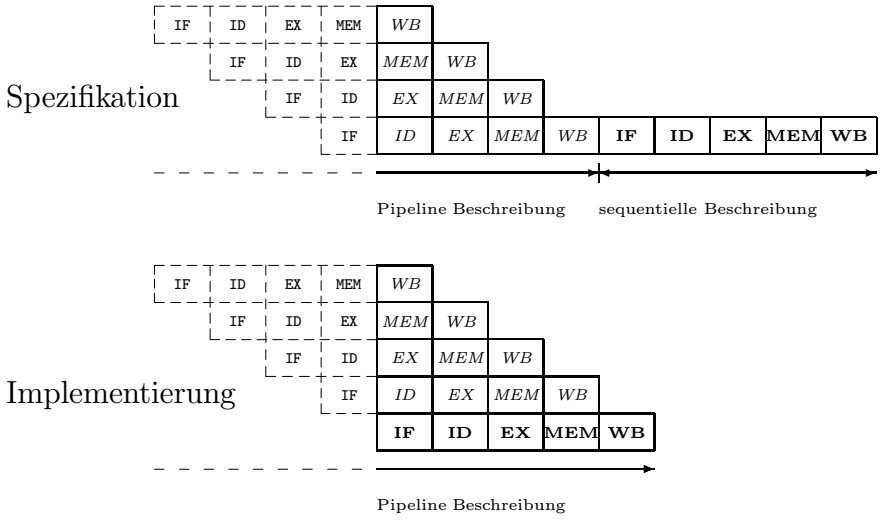


Abb. 5.37: Verifikation des DLX-Prozessors mit Pipelining

Beispiel 5.21 *Das Ablaufdiagramm in Abbildung 5.37 gibt ein Beispiel für die Verifikation eines Pipelinesystems. Die Befehle des DLX-Prozessors besitzen fünf verschiedene Befehlsphasen. Um die Pipeline zu verifizieren werden die beiden folgenden Sequenzen erzeugt:*

- *Spezifikation: Wird die Pipeline geleert, sind unter der Voraussetzung, daß die Pipeline nicht angehalten wird, vier Schritte erforderlich, bis alle Befehle abgearbeitet worden sind. Ein fünfter Befehl wird anschließend durch den sequentiellen Prozessor ausgeführt.*

- *Implementierung: Der Pipelinezustand realisiert alle fünf Befehlsphasen gleichzeitig. Durch das einmalige Durchlaufen des Pipelinezustands wird ein neuer, fünfter Befehl geladen. Vier Befehle befinden sich bereits in der Pipeline. Nach Laden des fünften Befehls wird die Pipeline geleert.*

In Abbildung 5.37 sind bei beiden Sequenzen die ersten vier Schritte für Spezifikation und Implementierung identisch, gestrichelt dargestellt, so daß ihre Äquivalenz nicht überprüft werden muß.

Sind die zwei finiten azyklischen Sequenzen berechnungsäquivalent, kann induktiv argumentiert werden, daß sich ein Programmablauf auf dem Pipelinesystem sukzessiv serialisieren läßt, d.h. in jedem Schritt wird ein weiterer Befehl auf dem sequentiellen Prozessor ausgeführt, bis dieser die Ausführung aller Instruktionen übernimmt. Ein beliebiges Programm wird somit dieselben Resultate auf dem sequentiellen Prozessor wie auf dem Pipelinesystem ergeben.

Beispiel 5.22 *In Abbildung 5.38 werden die Befehle sukzessiv serialisiert. Bereits ein Befehl wird auf dem sequentiellen Prozessor ausgeführt. Im nächsten Schritt werden drei, im folgenden vier und im letzten alle fünf sequentiell realisiert. Da induktiv argumentiert werden kann, ist es nicht erforderlich, die Korrektheit jedes dieser Übergänge zu zeigen. Es genügt, die Berechnungsäquivalenz der beiden in Abbildung 5.37 dargestellten Sequenzen nachzuweisen.*

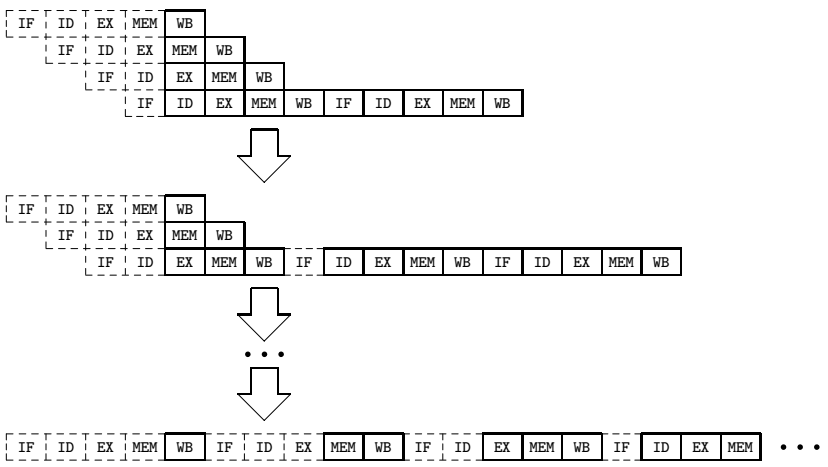


Abb. 5.38: Induktiver Korrektheitsbeweis

Der Äquivalenzprüfer vergleicht die Endwerte der Speicher und des Befehlszählers in Spezifikation und Implementierung paarweise. Die übrigen Register bleiben unbeachtet. Durch das Serialisieren des fünften Befehls in der Spezifikation ist es möglich, daß sich die Werte der übrigen Register unterscheiden. Die Pipeline-Register können z.B. verschiedene Werte speichern.

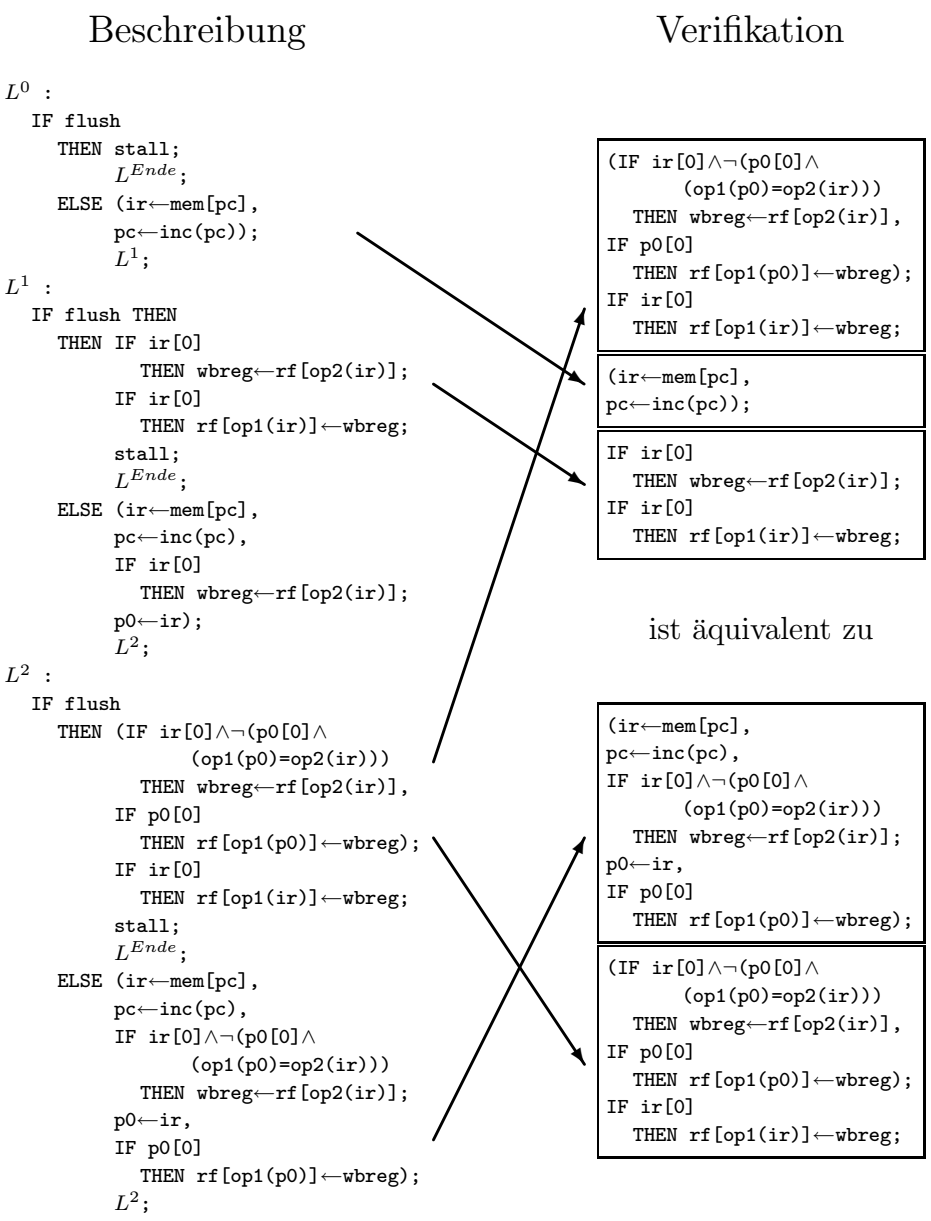


Abb. 5.39: Konstruktion der finiten Sequenzen

Beispiel 5.23 *Abbildung 5.39 zeigt, wie die zwei finiten azyklischen Sequenzen für das Beispiel der dreistufigen Pipeline aus Abschnitt 5.4.1 zusammengesetzt werden. Das Segment L2 beschreibt einerseits das Leeren der Pipeline und andererseits das Laden eines dritten Befehls* `IF` *parallel zu der* `Read`*- und* `Write`*-Phase der in der Pipeline befindlichen Instruktionen.*
Die phasenorientierte Beschreibung des sequentiellen Prozessors kann den Segmenten L0 und L1 entnommen werden. Dieser Teil der Beschreibung repräsentiert das sequentielle Ausführen eines Befehls.

In [BD94] werden Transitionsfunktionen definiert, die die Werte der Register und Speicher nach dem Leeren der Pipeline berechnen. Da die Verhaltensbeschreibung sowohl das Füllen als auch das Leeren der Pipeline beschreibt und eine phasenorientierte Kopie der sequentiellen Spezifikation beinhaltet, werden keine zusätzlichen Funktionen und Berechnungen benötigt. Der Korrektheitsbeweis kann automatisiert und in den Entwurfsfluß integriert werden.
Die Verhaltensbeschreibung einer Pipeline umfaßt mehrere Segmente. Jedes einzelne beschreibt eine bestimmte Kombination von Befehlsphasen in der Pipeline. Um die Implementierung der Pipeline zu verifizieren, muß der Korrektheitsbeweis für jedes dieser Segmente durchgeführt werden.

5.5.2 Beispiel einer dreistufigen Pipeline

Die Beschreibung der dreistufigen Pipeline (siehe Abschnitt 5.4.1) umfaßt drei Segmente, wobei jedes dieser Segmente zu verifizieren ist. Abbildung 5.39 stellt dar, wie die zwei zu vergleichenden finiten azyklischen Sequenzen für das Segment L2 erzeugt werden. Entsprechend ergeben sich die Sequenzen für die beiden anderen Segmente.

Zustand	Befehlsphasen	Anzahl der Pfade	Zeit
L0	IF	2	0,29 sec.
L1	IF, READ	5	0,32 sec.
L2	IF, READ, WRITE	13	0,33 sec.
			0,94 sec.

Tab. 5.17: Verifikation einer dreistufigen Pipeline

Die Tabelle 5.17 gibt einen Überblick über die verifizierten Zustände bzw. Segmente. Der Befehlssatz des Prozessors umfaßt nur einen Befehl, durch eine Verzweigung, die der Befehl ausführt, und durch Abhängigkeiten zwischen aufeinander folgenden Befehlen müssen aber mehrere Fallunterscheidungen getroffen werden. Aus diesem Grund sind bis zu 13 verschiedene Pfade symbolisch zu simulieren. Das Beispiel konnte innerhalb 1 Sekunde (300 MHz, SUN Ultra2) formal verifiziert werden.

5.5.3 Der DLX-Prozessor

Die Beschreibung des DLX-Prozessors mit Pipelining umfaßt neun Segmente. Die Segmente L0 bis L3 beschreiben das Füllen der Pipeline, L4 den Pipelinezustand, in dem alle Befehlsphasen parallel ausgeführt werden, und L5 bis L8 Befehlsfolgen, bei denen bestimmte Befehlsphasen fehlen, da durch Anhalten der Pipeline

oder unterschiedliche Längen der Befehle Instruktionen bereits terminiert sind.

Zustand	Befehlsphasen	Anzahl der Pfade	Zeit
L0	IF	8	0,40 sec.
L1	IF, ID	104	0,84 sec.
L2	IF, ID, EX	2.012	4,17 sec.
L3	IF, ID, EX, MEM	68.174	113,45 sec.
L4	IF, ID, EX, MEM, WB	1.370.198	2559,88 sec.
L5	IF, ID, WB	1.112	1,93 sec.
L6	IF, ID, MEM, WB	18.864	30,85 sec.
L7	IF, ID, EX, WB	44.360	67,69 sec.
L8	IF, ID, MEM	1.866	3,82 sec.
		1.506.698	46 min. 23 sec.

Tab. 5.18: Verifikation aller Zustände der DLX

Jedes dieser Segmente konnte erfolgreich überprüft werden. Details sind Tabelle 5.18 zu entnehmen. Die Verifikation des DLX-Prozessors mit Pipelining konnte in 46 Minuten (300 MHz, SUN Ultra2) durchgeführt werden. Wie Abschnitt 5.5.5 zu entnehmen ist, konnten durch die unabhängige Überprüfung einige Implementierungsfehler des Synthesewerkzeugs gefunden werden.

5.5.4 Verifikation des PIC-Prozessors

Der PIC-Prozessor aus Abschnitt 5.4.3 läßt sich unter Anwendung des Verfahrens nach Burch und Dill [BD94, EHR98, REH99, RHE99a] verifizieren. Sieben Register des PIC-Prozessors werden jedoch als Erweiterung der Registerbank `rf` modelliert. Aus diesem Grund werden in der Pipeline sehr viele Lese- und Schreibzugriffe auf `rf` ausgeführt. So stellt z.B. das Inkrementieren des Befehlszählers `pc`, modelliert als `rf[2]`, einen Lese- und einen Schreibzugriff dar. Da in jeder Befehlsphase auf die Register der Registerbank zugegriffen wird, erfolgen viele Zugriffe gleichzeitig. In einigen Fällen verändert sich sogar die Reihenfolge der Zugriffe.

Beispiel 5.24 *Abbildung 5.40 gibt ein Beispiel für eine veränderte Reihenfolge der Lese- und Schreibzugriffe. In der Spezifikation wird auf den Speicher* `mem` *zuerst geschrieben, um anschließend einen Wert auszulesen. In der Implementierung dreht sich die Reihenfolge um, da* `x` *den Wert* `val` *bei Adressengleichheit* `adrV=adrX` *durch Forwarding übernimmt. Zusätzlich werden in der Implementierung die zwei Schreiboperationen auf die Registerbank* `rf` *vertauscht. Gilt* `adrA=adrB`*, überschreibt* `rf[adrB]←b` *in der Spezifikation* `rf[adrA]←a`*. In der Implementierung wird* `a` *nicht abgespeichert, so daß eine Schreiboperation*

entfällt. Die Berechnungsäquivalenz der beiden Sequenzen läßt sich zeigen, indem gleiche Speicherzustände [RHE99b] *für* `mem` *und* `rf` *identifiziert werden.*

Spezifikation

```
rf[adrA]←a;
rf[adrB]←b;
mem[adrV]←val;
x←mem[adrX];
z←x+rf[adrZ];
```

Implementierung

```
(rf[adrB]←b,
x←mem[adrX]);
(if ¬(adrA=adrB)
   then rf[adrA]←a,
mem[adrV]←val);
(if adrV=adrX
   then z←val+rf[adrZ]
   else z←x+rf[adrZ]);
```

Abb. 5.40: Äquivalente Lese- und Schreibzugriffe durch Forwarding

Die Äquivalenz von Speicheroperationen wird durch Identifizieren äquivalenter Speicherzustände in den zu vergleichenden Sequenzen nachgewiesen. Da jedoch nur Schreibzugriffe einen Speicher verändern, genügt es, äquivalente Schreibzugriffe herauszusuchen [RHE99b]. Es ist zu beachten, daß sich die Reihenfolge der Schreibzugriffe sowie auch deren Anzahl verändern kann. Zu einer veränderten Reihenfolge der Zugriffe kommt es, wenn im Pipelinesystem in unterschiedlichen Phasen geschrieben wird. Sind Adressen gleich und werden Werte überschrieben, ändert sich die Anzahl der Zugriffe. Auf der Basis äquivalenter Speicherzustände kann auch die Äquivalenz von Leseoperationen nachgewiesen werden.
Tabelle 5.19 gibt einen Überblick über die Verifikation der in Abschnitt 5.4.3 dargestellten Entwürfe. Während in Spalte zwei die Anzahl der bei der symbolischen Simulation durchlaufenen Pfade zu finden ist, gibt Spalte drei die Anzahl der logisch unmöglichen Pfaden an, die der Äquivalenzprüfer [REH99] durchläuft. Spalte vier stellt dar, wie oft sich Schreiboperationen überschreiben, Spalte fünf zeigt, wie oft sich die Reihenfolge der Speicherzugriffe verändert hat, und Spalte sechs ist die Zeit zu entnehmen, die für die Verifikation benötigt wird.

PIC	Anzahl Pfade	logisch unmögl. Pfade	Schreibzugriffe		Zeit für die Verifikation
			überschrieben	veränderte Reihenfolge	
1	58.135	18	306	25	2 min 16 sec
2	49.345	18	277	25	1 min 55 sec
3	193.864	123	3.455	3.378	7 min 30 sec
4	202.565	124	1.566	1.566	10 min 5 sec

Tab. 5.19: Verifikation der PIC-Prozessoren

Die Beschreibung (PIC 4) modelliert den Befehlszähler als `rf[2]` und das Statusregister als `rf[3]`. Trotz der höchsten Pfadanzahl treten in dieser Beschreibung weniger Komplikationen auf als in (PIC 3). Da in (PIC 4) mehr Datenabhängigkeiten auftreten, muß die Pipeline öfter angehalten werden, so daß weniger über-

schriebene Schreiboperationen existieren und sich deren Reihenfolge weniger oft verändert.

5.5.5 Gefundene Implementierungsfehler

Durch die Verifikation der Ergebnisse konnten mehrere Fehler in der Implementierung des Transformationswerkzeugs gefunden werden, die jedoch nicht die Korrektheit der Transformationen in Frage stellten. Im folgenden wird ein bei der Verifikation des DLX-Prozessors mit Pipelining gefundener Fehler dargestellt, der zeigt, wie schwierig es ist, Fehler zu finden und wie wichtig somit die automatisierte formale Verifikation von Syntheseergebnissen ist.

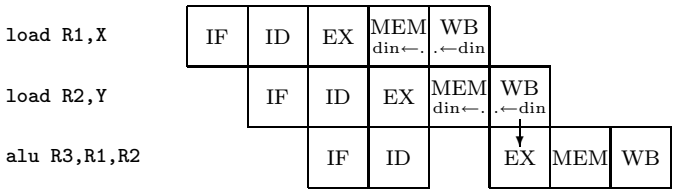


Abb. 5.41: Befehlsfolge load-load-alu

In dem Fall, daß die beiden Operanden eines `alu`-Befehls durch zwei unmittelbar vorausgehende `load`-Befehle geladen wurden, arbeitete die Pipeline nicht korrekt. Die Bearbeitung des `alu`-Befehls muß, da Adressengleichheit zwischen dem `alu`- und dem vorangehenden `load`-Befehl besteht, für einen Schritt ausgesetzt werden. In der MEM-Phase lädt die `load`-Instruktion den Wert aus dem Speicher und speichert ihn in `din`, während in der WB-Phase das Ergebnis in ein Register der Registerbank abgespeichert wird. Parallel zu der WB-Phase des `load`-Befehls führt die `alu`-Instruktion die EX-Phase aus, so daß durch Forwarding der vom `load` geladene Operand übernommen werden kann. Der erste `load`-Befehl terminiert jedoch während des Anhaltens der Pipeline, so daß sein Ergebnis nicht durch Forwarding in der EX-Phase des `alu`-Befehls berücksichtigt werden kann. Der nach `din` geladene Wert wird von dem nachfolgenden `load`-Befehl überschrieben. Somit konnte nur ein Operand des `alu`-Befehls korrekt bestimmt werden. Abbildung 5.41 stellt das zugehörige Ablaufdiagramm dar. Das Forwarding eines Wertes ist durch einen vertikalen Pfeil angedeutet.
Der Fehler läßt sich durch Wiederholen der ID-Phase des `alu`-Befehls beheben. Wie in Abbildung 5.42 dargestellt, kann der Wert, den der erste `load`-Befehl lädt, dann bereits in der zweiten ID-Phase übernommen werden. Fehler können durch das Wiederholen der ID-Phase nicht auftreten, da entweder ein veralteter Wert von einem aktuelleren Wert überschrieben oder der gleiche Wert erneut geladen wird.

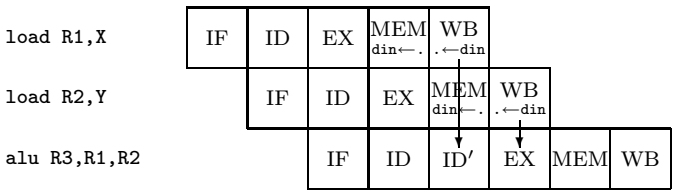


Abb. 5.42: Wiederholen der ID-Phase

Eine zweite Möglichkeit besteht darin, ein zusätzliches Pipeline-Register $\mathtt{p}_d$ einzuführen, das den durch das erste `load` geladenen Wert zwischenspeichert. Das Ablaufdiagramm ist Abbildung 5.43 zu entnehmen.
Während des Transformationsprozesses wird aufgrund von Datenabhängigkeiten das Einführen von Pipeline-Registern notwendig. Der Implementierungsfehler bestand darin, daß diese Datenabhängigkeiten nicht entdeckt und daher kein Pipeline-Register eingeführt worden ist. Wird das Verfahren dagegen korrekt implementiert, wird während des Transformationsprozesses ein Pipeline-Register eingeführt, was zur Lösung des dargestellten Konflikts führt.

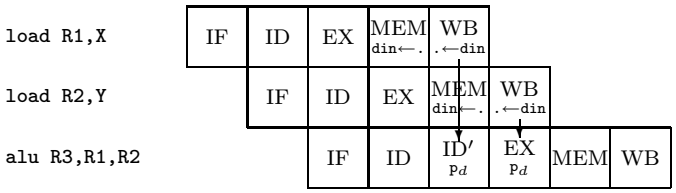


Abb. 5.43: Einführen eines Pipeline-Registers

5.6 Zusammenfassung

Die experimentellen Ergebnisse zeigen, daß durch Anwenden einer Folge korrektheitserhaltender Transformationen aus einer sequentiellen Prozessorbeschreibung ein Pipelinesystem abgeleitet werden kann. Das Verfahren parallelisiert sukzessiv die Befehlsphasen. Befehlsphasen werden unterschiedlichen Kontrollschritten zugeordnet, falls die Datenabhängigkeiten unter Einhaltung der Ressourcenbeschränkungen nicht gelöst werden können.
Ist es nicht möglich, Befehlsphasen zu parallelisieren, werden die Bedingungen, unter denen der Konflikt auftritt, herausgesucht. Diese werden dann verwendet, um eine neue Verzweigung einzuführen. Ist deren Bedingung erfüllt, können die Operationen ohne Verletzen der Ressourcenbeschränkungen parallel realisiert werden, da keine Konflikte entstehen oder Techniken wie Forwarding oder

Einführen von Pipeline-Registern diese verhindern. Andernfalls werden Leerschritte eingefügt, so daß die Operationen verzögert ausgeführt werden. Das Einführen neuer Verzweigungen erfordert die Anwendung von Vereinfachungstechniken, um Redundanzen und logisch unmögliche Pfade zu eliminieren (siehe Kapitel 7).
Die fall-basierte Einplanung plant eine dynamische Pipeline ein. Dabei werden alle Befehlsabfolgen ermittelt, bei deren Ausführung in der Pipeline Konflikte aufgrund von Datenabhängigkeiten oder Verletzungen der Ressourcenvorgaben bestehen. Durch die Kombination mit Methoden der Post-Synthese-Verifikation wird eine automatisierte Synthese möglich, die einerseits konzeptionell korrekt ist, da sie auf mathematischen Konzepten beruht, und die sich andererseits robust gegen Implementierungsfehler zeigt, weil die Äquivalenz von Spezifikation und eingeplanter Beschreibung durch eine symbolische Simulation nachgewiesen wird.
Das *TUD Transformationswerkzeug (TUDT)* erzeugt eine Verhaltensbeschreibung der Pipeline. Eine strukturelle Repräsentation kann bisher noch nicht transformativ gewonnen werden, sondern wird mit Hilfe des *Synopsys Design Compiler* [Syn98a] erzeugt.

Kapitel 6

Verifikation von Einplanungsverfahren

6.1 Einleitung

Transformative Ansätze werden nicht nur im Bereich der formalen Synthese verfolgt, sondern kommen auch im Bereich der Post-Synthese-Verifikation zum Einsatz. Die Äquivalenz von zwei Beschreibungen ist gegeben, wenn eine Folge korrektheitserhaltender Transformationen existiert, die die eine in die andere Beschreibung umwandelt. Der Nachweis der Transformationsäquivalenz erfolgt, indem die Beschreibungen durch eine Reihe von Transformationen angeglichen werden, so daß eine Bisimulationsrelation zwischen den Segmenten der einen und denen der anderen Beschreibung aufgestellt werden kann [EHR99]. Die Bisimulation ist ein Konzept, das z.B. im Bereich der Prozeßkalküle [Hoa85, Mil89] Verwendung findet.
Das Verfahren der transformativen Verifikation setzt voraus, daß eine *gewisse Ähnlichkeit* zwischen den Beschreibungen besteht. Werden Datenoperationen inhaltlich verändert, indem z.B. eine Shift-Operation eine Multiplikation ersetzt, so ist der Nachweis auf Basis des vorgestellten transformativen Ansatzes nicht mehr möglich.
Strukturelle Ähnlichkeiten erleichtern es, die Äquivalenz von zwei Schaltungen zu zeigen. In [KK97] werden z.B. auf Gatterebene äquivalente Teilschaltungen identifiziert und die Schaltungen schrittweise angeglichen. Die Äquivalenz von endlichen Automaten wird in [vE98] nachgewiesen, indem unter Anwendung von Retiminig-Transformationen eine Fixpunktberechnung durchgeführt wird. Die strukturellen Ähnlichkeiten der Automaten werden ausgenutzt, indem in den Automaten gleiche Zustandsfunktionen identifiziert werden, so daß nicht der gesamte Zustandsraum des zugehörigen Produktautomaten traversiert werden muß.

Einplanungsverfahren legen die Ausführungszeitpunkte von Operationen fest. Berücksichtigt werden dabei Datenabhängigkeiten, die Verfügbarkeit von funktionalen Einheiten usw. Im allgemeinen beachten solche Verfahren nicht die Inhalte der in den Ausdrücken verwendeten Funktionen, so daß die Datenoperationen bis auf das Einführen zusätzlicher Register unverändert bleiben.
Die Ergebnisse von Verfahren für azyklische Beschreibungen wie z.B. *List-Scheduling* [MLD92] oder *Force directed Scheduling* [PK89a] lassen sich durch Werkzeuge wie z.B. *SVC (Stanford Validity Checker)* [JDB95, BDL96] verifizieren. Es wird die Äquivalenz von Formeln gezeigt, die die Endwerte der Variablen durch die Anfangswerte ausdrücken. Da die Ausdrücke extrem groß werden können, bietet sich in vielen Fällen die symbolische Simulation [REH99] zur Verifikation an. Die Korrektheit der Ergebnisse von Verfahren, die wie z.B. *As Fast As Possible* [Cam91], *Dynamic Loop Scheduling* [ROJ94], *Pipelined Path Scheduling* [RJ95b] oder *die fall-basierte Einplanung* [Hin98a, HER99, HRE99] zyklische Beschreibungen einplanen, läßt sich jedoch mit solchen Verfahren nicht nachweisen.

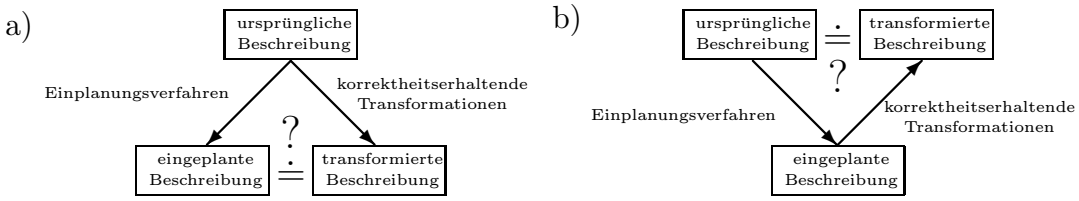


Abb. 6.1: Verifikation durch Anwendung von Transformationen

Wird ein transformativer Ansatz gewählt, kann das Einplanungsergebnis durch Aufstellen einer Bisimulation verifiziert werden. Korrektheitserhaltende Transformationen werden solange angewendet, bis sich ursprüngliche und eingeplante Beschreibung gleichen. Die Transformationen können entweder auf die ursprüngliche Beschreibung, wie in Abbildung 6.1(a) dargestellt, oder auf das Ergebnis der Einplanung 6.1(b) angewendet werden. Ebenso ist es möglich, abwechselnd die Beschreibungen zu verändern. Im folgenden wird entsprechend Abbildung 6.1(a) verfahren.

Beispiel 6.1 *Abbildung 6.2 stellt zwei transformationsäquivalente Beschreibungen dar, die [RJ95a] entnommen sind. Beide Beschreibungen berechnen* $(a \cdot b)$ *mod n. Die rechte Beschreibung ergibt sich aus der linken nach Ausführung eines Dynamic Loop Scheduling (DLS) [ROJ94]. Verifizieren läßt sich die Einplanung, indem eine der beiden Beschreibungen derart transformiert wird, daß sie der anderen Beschreibung entspricht. Es zeigt sich, daß die beiden Beschreibungen äquivalent sind.*

Während Abschnitt 6.2 das Verfahren der transformativen Verifikation im Detail darstellt, gibt Abschnitt 6.3 experimentelle Ergebnisse. Eine Zusammenfassung und ein Ausblick beenden das Kapitel.

$\mathcal{B}^{mod} = (\{L^0, L^1, L^2, L^3, L^4\}, L^0, \{a, b, n, i, s, sout\})$ mit

```
L0 :
  (s←0,i←0);
  L1;
L1 :
  IF i≤15
    THEN IF odd(b)
           THEN s←s+a;
                L2;
           ELSE stall;
                L3;
    ELSE sout←s;
         LEnde;
L2 :
  IF s>n
    THEN s←s−n;
         L3;
    ELSE stall;
         L3;
L3 :
  (i←i+1,
  b←b/2,
  a←a∗2);
  L4
L4 :
  IF a>n
    THEN a←a−n;
         L1;
    ELSE stall;
         L1;
```

$\mathcal{B}^{dls} = (\{S^0, S^1, S^2, S^3, S^4\}, S^0, \{a, b, n, i, s, sout\})$ mit

```
S0 :
  (s←0,i←0);
  S1;
S1 :
  IF i≤15
    THEN IF odd(b)
           THEN s←s+a;
                S2;
           ELSE (i←i+1,
                b←b/2,
                a←a∗2);
                S3;
    ELSE sout←s;
         SEnde;
S2 :
  IF s>n
    THEN (s←s−n,
         i←i+1,
         b←b/2,
         a←a∗2);
         S3;
    ELSE (i←i+1,
         b←b/2,
         a←a∗2);
         S3;
S4 :
  s←s+a;
  S2;
S5 :
  a←a∗2;
  S3;
S3 :
  IF a>n
    THEN IF i≤15
           THEN IF odd(b)
                  THEN a←a−n;
                       S4;
                  ELSE (a←a−n,
                       i←i+1,
                       b←b/2);
                       S5;
           ELSE (a←a−n,
                sout←s);
                SEnde;
    ELSE IF i≤15
           THEN IF odd(b)
                  THEN s←s+a;
                       S2;
                  ELSE (i←i+1,
                       b←b/2,
                       a←a∗2);
                       S3;
           ELSE sout←s;
                SEnde;
```

Abb. 6.2: Zwei transformationsäquivalente Beschreibungen

6.2 Transformative Verifikation

Die Transformationsäquivalenz zweier Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ ist gegeben, wenn eine Folge von Transformationen existiert, die $\mathcal{B}$ in $\mathcal{B}'$ umformt. $\mathcal{B}$ entspricht $\mathcal{B}'$, falls gezeigt werden kann, daß eine Bisimulation $\approx$ zwischen $\mathcal{B}$ und $\mathcal{B}'$ existiert.

Definition 6.1 (Bisimulation)
Zwischen zwei Beschreibungen $\mathcal{B}$ und $\mathcal{B}'$ besteht genau dann eine Bisimulation $\mathcal{B} \approx \mathcal{B}'$, wenn zu jedem Segment $S \in \mathcal{S}_{\mathcal{B}}$ ein bisimulares Segment $S' \in \mathcal{S}_{\mathcal{B}'}$ existiert und umgekehrt: $(S, S') \in\approx$. $(S, S') \in\approx$ ist für zwei Segmente S und S' gegeben, falls

- *in S und S' die gleichen Datenoperationen ausgeführt werden und*
- *nach Ausführung von S und S' bisimulare Segmente aufgerufen werden.*

Beispiel 6.2 *Jede normalisierte LLS-Beschreibung kann als erweiterter endlicher Automat dargestellt werden. Die beiden in Abbildung 6.3 dargestellten, er-*

weiterten Automaten enthalten die bisimularen Zustände L^a und S^a sowie L^b und S^b, da die gleichen Datenoperationen ausgeführt werden und die nachfolgenden Zustände ebenfalls bisimular sind.
Für den Zustand L^c existiert in dem rechten Automaten kein bisimularer Zustand. Sowohl S^a als auch S^b führen eine Datenoperation aus. L^c verzweigt jedoch nur nach L^a bzw. L^b und verändert keine Daten. Somit sind die Automaten nicht bisimular.

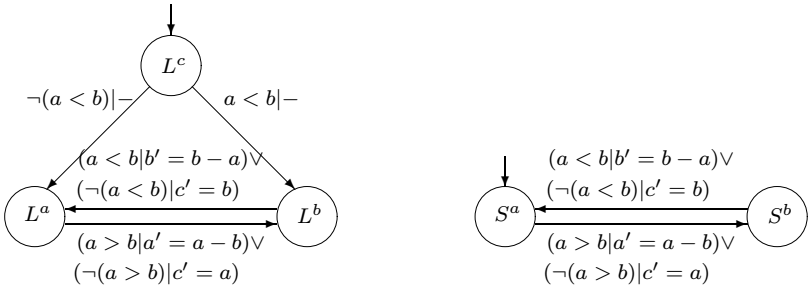


Abb. 6.3: Zwei erweiterte endliche Automaten zur Bestimmung des GGT

Anhand der Darstellung als Automat ist es nicht möglich, die Äquivalenz zweier Beschreibungen zu überprüfen. Die Automaten sind erweitert, d.h. an den Kanten werden boolesche Ausdrücke als Bedingungen für einen Übergang und auszuführende Datenoperationen notiert. Somit besteht keine Möglichkeit, einen eindeutigen, minimalen Automaten [HU94] zu erzeugen.
Die Transformationsäquivalenz zweier Beschreibungen läßt sich beweisen, indem die Bisimulation der Anfangssegmente gezeigt wird. Um die Anfangssegmente als bisimular zu identifizieren, muß in Breitensuche die Bisimulation der nachfolgenden Segmente nachgewiesen werden. Führen zwei Segmente ungleiche Datenoperationen aus, wird eines der beiden Segmente durch Anwendung von Transformationen derart verändert, daß die Bisimulation gezeigt werden kann. Ist dies nicht möglich, sind die Segmente und somit auch die Beschreibungen nicht bisimular. Formale Transformationen (siehe Abschnitt 4.6) kommen zur Anwendung, um den Kontrollfluß zu modifizieren, neue Segmente zu erzeugen oder die Segmentübergänge zu verändern. So werden z.B. Anweisungen in if-then-else-Strukturen hineingezogen oder Ausgangsmarken durch den Körper des zugehörigen Segments ersetzt. Kontext-abhängige Transformationen (siehe Abschnitt 4.7), die nur auf azyklische Sequenzen von Anweisungen angewendet werden können, werden verwendet, um Anweisungen zu parallelisieren oder zu serialisieren. Sie verändern die zeitlichen Eigenschaften einer Beschreibung.

Beispiel 6.3 *Die in Abbildung 6.2 dargestellten Beschreibungen sind transformationsäquivalent. $(L^1, S^1) \in \approx$ kann erst nach Anwendung mehrerer Transformationen nachgewiesen werden. Abbildung 6.4(a) ist das unveränderte Segment*

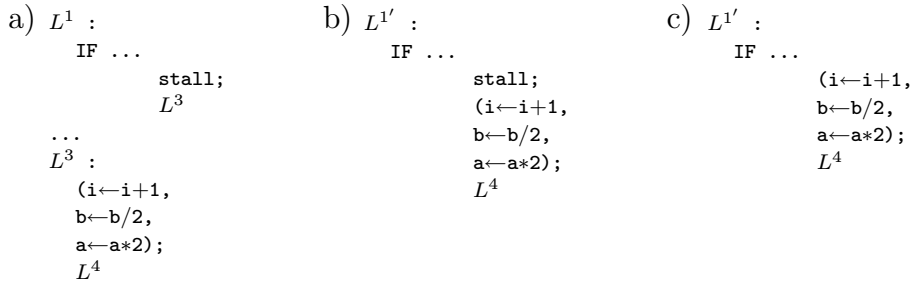


Abb. 6.4: Beispiel für die Anwendung von Transformationen

L^1 zu entnehmen. Zunächst wird die Ausgangsmarke L^3, wie in Abbildung 6.4(b) dargestellt, durch den Körper des zugehörigen Segments ersetzt. Das transformierte Segment wird als $L^{1'}$ bezeichnet. Werden `i←i+1`, `b←b/2` *und* `a←a∗2` *einen Zeitschritt vorgezogen, entfällt der Leerschritt, so daß $L^{1'}$ und S^1) $\in\approx$ die gleichen Datenoperationen ausführen.*

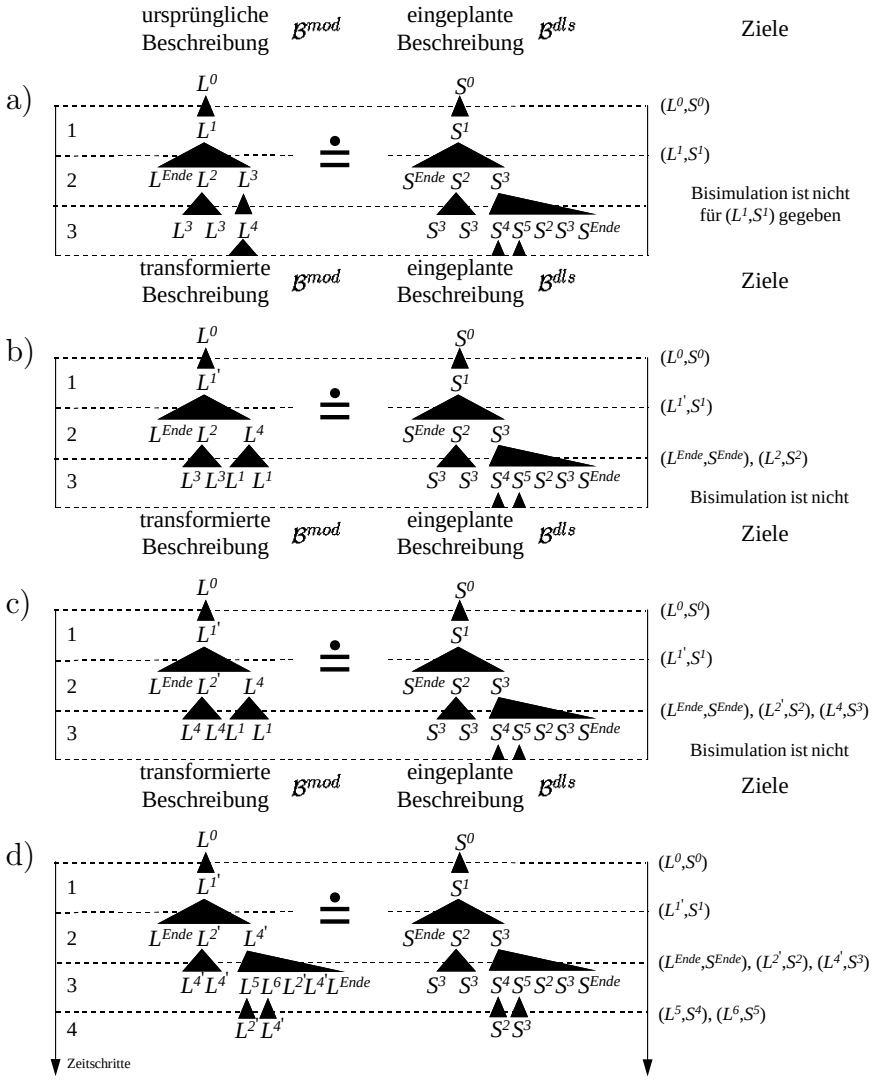


Abb. 6.5: Aufstellen einer Bisimulation durch Transformation

Abbildung 6.2 sind zwei transformationsäquivalente Beschreibungen $\mathcal{B}^{mod}$ und $\mathcal{B}^{dls}$ zu entnehmen. $\mathcal{B}^{dls}$ ist das Ergebnis der Einplanung von $\mathcal{B}^{mod}$. Durch den Nachweis der Transformationsäquivalenz der beiden Beschreibungen kann gezeigt werden, daß das Einplanungsverfahren korrekt gearbeitet hat. Über die Güte des Ergebnisses wird allerdings keine Aussage gemacht. Abbildung 6.5 gibt einen Überblick über den Nachweis der Transformationsäquivalenz.
Da die Startsegmente L^0 und S^0 gleiche Datenoperationen ausführen, ist die Bisimulation $(L^0, S^0) \in \approx$ gegeben, wenn zwischen den beiden nachfolgenden Segmenten L^1 und S^1 ebenfalls eine Bisimulation besteht. Wie in Abbildung 6.5(a) dargestellt, gilt $(L^1, S^1) \notin \approx$, weil unterschiedliche Operationen ausgeführt werden. Während L^1 einen Leerschritt ausführt, werden in S^1 die Zuweisungen `i←i+1`, `b←b/2` und `a←a*2` realisiert. Durch Anwenden von Transformationen auf L^1 wird die Ausgangsmarke L^3 durch den Körper des entsprechenden Segments ersetzt und der Leerschritt entfernt. Das Segment wird im folgenden als $L^{1'}$ bezeichnet, um zwischen ursprünglichem und transformiertem Segment unterscheiden zu können. Es gilt $(L^{1'}, S^1) \in \approx$ und somit auch $(L^0, S^0) \in \approx$, falls $(L^2, S^2) \in \approx$ und $(L^4, S^3) \in \approx$ erfüllt sind. L^{Ende} und S^{Ende} terminieren die Ausführung einer Beschreibung, so daß $(L^{Ende}, S^{Ende}) \in \approx$ gegeben ist. Wie Abbildung 6.5(b) und 6.5(c) zu entnehmen ist, sind $(L^2, S^2) \in \approx$ und $(L^4, S^3) \in \approx$ nicht erfüllt. Erst nach Anwendung weiterer Transformationen ergibt sich

$$\approx = \{(L^0, S^0), (L^{1'}, S^1), (L^{Ende}, S^{Ende}), (L^{2'}, S^2), (L^{4'}, S^3), (L^5, S^4), (L^6, S^5)\}.$$

Abbildung 6.5(d) stellt das Ergebnis dar.
Für kontext-abhängige Transformationen existieren keine inversen Transformationen (siehe Abschnitt 4.7). Werden kontext-abhängige Transformationen angewendet, können die durch Forwarding ersetzten Ausdrücke, die eingeführten Pipeline-Register und die eliminierten Anweisungen nicht mehr **eindeutig** bestimmt werden. Aus diesem Grund kann das vorgestellte Verfahren daran scheitern, daß eine Beschreibung, in der solche Techniken angewendet werden, nicht an eine Beschreibung, in der die Anweisungen nicht parallel ausgeführt werden, angeglichen werden kann.

a)
```
q←p;
IF q
  THEN (a←d,b←e,c←<...>+a);
  ELSE ...
```

b)
```
IF p
    THEN (a←<...>,b←a);
         (a←d,b←e,c←a+b;
    ELSE ...
```

Abb. 6.6: Probleme bei dem Nachweis der Transformationsäquivalenz

Beispiel 6.4 *In Abbildung 6.6 wird ein Beispiel dargestellt, bei dem es schwierig wird, die Äquivalenz mit dem beschriebenen Verfahren nachzuweisen. Soll die Beschreibung in Abbildung 6.6(a) durch Anwendung von Transformationen an 6.6(b) angeglichen werden, ist ein eingeführtes Pipeline-Register* `q` *zu eliminieren und die durch Forwarding erzeugte Zuweisung* `c←<...>+a` *auf* `a←<...>`, `b←a`

und `c←a+b` *zurückzuführen.*

Da die Anwendung der kontext-abhängigen Transformationen auf finite azyklische Sequenzen von Anweisungen begrenzt ist, können zum Nachweis der Äquivalenz Methoden der Post-Synthese-Verifikation eingesetzt werden. Der Äquivalenzbeweis kann z.B. durch eine symbolische Simulation [REH99, RHE99a] erfolgen. Derzeit werden solche Verfahren noch nicht eingesetzt. Das Verfahren soll aber derart erweitert werden, daß Sequenzen, die nicht durch kontext-abhängige Transformationen eindeutig ineinander überführbar sind, mit Methoden der Post-Synthese-Verifikation auf Äquivalenz überprüft werden.

6.3 Experimentelle Ergebnisse

Der Nachweis der Transformationsäquivalenz konnte für mehrere in der Literatur angegebene Beispiele erbracht werden. Tabelle 6.1 stellt die Ergebnisse dar. Die Messungen wurden auf einer SUN Ultra2 300 MHz durchgeführt. PREFETCH ist ein Beispiel des *As Fast As Possible* Verfahrens aus [Cam91]. MODULO berechnet $(a \cdot b)$ *mod* n und wurde [RJ95a] entnommen. Zur Einplanung wurden die Verfahren *As Fast As Possible*, *Dynamic Loop Scheduling* und *Pipelined Path Based Scheduling* verwendet.

Entwurf	Einplanungs-verfahren	Laufzeit
PREFETCH	AFAP	0,06 sec
MODULO	AFAP	0,10 sec
	DLS	0,19 sec
	PPS	0,10 sec
SHEWA	feasible	1,35 sec
	optimal	1,18 sec
3STAGE	3-stufige Pipeline	0,08 sec

Tab. 6.1: Verifikation von Einplanungsergebnissen

Kontext-abhängige Transformationen gestatten es, Anweisungen zu parallelisieren. Aus diesem Grund können auch Ergebnisse von Einplanungsverfahren verifiziert werden, die zusätzliche Pipeline-Register einführen oder Techniken wie Forwarding anwenden. Das SHEWA-Beispiel wurde von dem *Shewa System* [PP88] unter Verwendung des *Forward Feasible Scheduling* Verfahrens und des *Forward Optimal Scheduling* erzeugt. Auch eine dreistufige Pipeline 3STAGE, die Abschnitt 5.4.1 entnommen wurde, konnte verifiziert werden. Sie wurde unter Verwendung der fall-basierten Einplanung [HER99] aus einer sequentiellen Beschreibung abgeleitet.

6.4 Zusammenfassung und Ausblick

Transformative Ansätze können auch im Bereich der Verifikation eingesetzt werden. Das vorgestellte Verfahren weist die Transformationsäquivalenz zweier Beschreibungen nach. Unter Anwendung einer Folge von Transformationen wird die eine Beschreibung der anderen angeglichen. Die Beschreibungen sind transformationsäquivalent, wenn sich eine Bisimulationsrelation zwischen den Segmenten der beiden Beschreibungen aufstellen läßt.
Experimentelle Ergebnisse zeigen, daß dieses Verfahren zur automatisierten Verifikation von Ergebnissen moderner zyklischer Einplanungsverfahren verwendet werden kann. Neben formalen werden auch kontext-abhängige Transformationen verwendet, so daß auch Ergebnisse von Einplanungsverfahren verifiziert werden können, die zusätzliche Pipeline-Register einführen oder Techniken wie Forwarding anwenden.
Da kontext-abhängige Transformationen nicht eindeutig invertiert werden können, ist für bestimmte Beschreibungen die Transformationsäquivalenz nicht nachweisbar. Es ist geplant, das Verfahren, das bisher auf die Anwendung von Transformationen beschränkt ist, derart zu erweitern, daß auch ein Äquivalenzprüfer zum Einsatz kommt, mit dem die Berechnungsäquivalenz von Sequenzen überprüft werden kann, die nicht ineinander transformierbar sind.

Kapitel 7

Vereinfachung algorithmischer Beschreibungen

7.1 Einleitung

Das Vereinfachen azyklischer Sequenzen bedingter Zuweisungen ist aufgrund der Komplexität der Bedingungen und sequentieller Abhängigkeiten der Anweisungen schwierig. Solche Strukturen treten typischerweise in Verhaltensbeschreibungen von Prozessoren auf, insbesondere, wenn sie das Ergebnis einer automatischen Einplanung sind.
Vereinfachungstechniken erlauben es, durch Verringern der Komplexität der Kontrollogik und Eliminieren logisch unmöglicher Pfade Kosten zu reduzieren und/ oder eine Geschwindigkeitsverbesserung zu erzielen. Im folgenden wird ein Verfahren beschrieben, das algorithmische Beschreibungen unabhängig von dem jeweiligen Syntheseschritt vereinfacht. Bereits in [Ber91] wurde ein Ansatz zum Beseitigen logisch unmöglicher Pfade vorgestellt, der jedoch kein unabhängiges Verfahren zur Vereinfachung einer Beschreibung darstellt, sondern im Verlauf einer pfadbasierten Einplanung [Cam91] durchgeführt wird.
Das dargestellte Verfahren wurde entwickelt, um einerseits eine automatisierte Einplanung für Prozessoren mit Pipelining zu ermöglichen, da in einigen Fällen die Durchführung ohne Anwendung von Minimierungstechniken an der zu großen Rechenzeit scheitert [Hin98a, HER99]. Andererseits sollte aber auch die Verifikation durch Eliminieren logisch unmöglicher Pfade und Vereinfachen von Kontrollstrukturen erleichtert werden.
Die Vereinfachung sequentieller Beschreibungen wird dadurch erschwert, daß die auszuführenden Operationen von komplexen booleschen Ausdrücken, in denen sowohl Bitvektor- als auch arithmetische Operationen möglich sind, bedingt werden können. Um eine effiziente Vereinfachung gewährleisten zu können, müssen desweiteren sequentielle Abhängigkeiten zwischen Bedingungen und Zuweisun-

gen gelöst werden. Abbildung 7.1 gibt ein einfaches Beispiel.

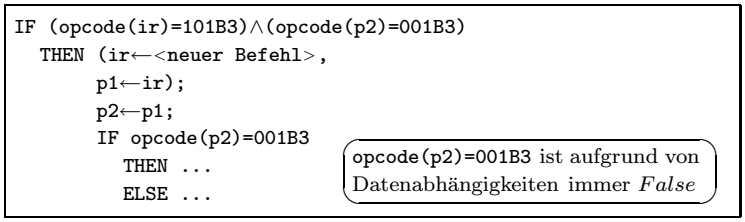


Abb. 7.1: Datenabhängigkeiten zwischen Bedingungen und Zuweisungen

Da sowohl sequentielle Abhängigkeiten bestehen als auch für eine effektive Minimierung jeder Pfad einer Beschreibung und jede auf einem Pfad gültige Bedingung berücksichtigt werden müssen, ist aufgrund des Berechnungsaufwandes eine exakte Lösung nicht möglich. Die verwendeten Heuristiken finden durch Einbeziehung des wechselseitigen Ausschlusses und der Implikation von Bedingungen, durch Berücksichtigen aller auf einem Pfad vorausgegangenen Bedingungen und durch Lösen sequentieller Datenabhängigkeiten logisch unmögliche Pfade und ermöglichen eine effiziente Vereinfachung von Bedingungen.
Während Abschnitt 7.2 einen Überblick über das Verfahren gibt, zeigt Abschnitt 7.3, daß korrektheitserhaltende Transformationen zur Vereinfachung von if-then-else Strukturen verwendet werden können. Vereinfachungstechniken, die auf *binären Entscheidungsdiagrammen* [Bry86] und algebraischen Gesetzen beruhen, werden in Abschnitt 7.4 vorgestellt. Verfeinerungstechniken, die in Abschnitt 7.5 beschrieben werden, gestatten es, azyklische Sequenzen von Anweisungen zu vereinfachen, indem Datenabhängigkeiten durch Methoden wie *Forwarding* oder *boolesche Reduktion* gelöst werden. Der Algorithmus *Simplify* wird in Abschnitt 7.6 dargestellt. Experimentelle Ergebnisse und eine Zusammenfassung beschließen das Kapitel.

7.2 Überblick

Korrektheitserhaltende Transformationen erlauben es, Strukturen von if-then-else-Ausdrücken zu vereinfachen. Mehrfach auftretende, identische Zweige von if-then-else-Anweisungen können mit Hilfe von formalen Transformationen eliminiert werden. Durch das Anwenden dieser Transformationen steigt zwar die Komplexität der Bedingungen der if-then-else-Ausdrücke, doch lassen sich diese häufig vereinfachen.
Um sequentielle Anweisungen effizient zu vereinfachen, müssen alle Pfade und alle auf einem Pfad gültigen Bedingungen berücksichtigt werden. Jede auf einem Pfad gefundene Bedingung kann den Bereich nachfolgender Ausdrücke einschränken. Im folgenden werden alle Bedingungen, die auf einem Pfad vorausgehen und die

die Gültigkeit der nachfolgenden Ausdrücke einschränken, als *vorausgehende Bedingungen* bezeichnet. Entsprechend werden Bedingungen, die auf einem Pfad nachgestellt sind, *nachfolgende Bedingungen* genannt.
Neben den auf einem Pfad gültigen Bedingungen wird der *wechselseitige Ausschluß* von Prädikaten beachtet. Berücksichtigt werden Vergleiche von Vektoren und Konstanten, da solche Ausdrücke in der Kontrollogik oft zu finden sind und ihr wechselseitiger Ausschluß leicht entdeckt werden kann. Zwei Ausdrücke schließen sich wechselseitig aus, wenn die gleiche Variable mit unterschiedlichen Konstanten verglichen wird, wie z.B. `ir[0:2]=010B3` und `ir[1:4]=1101B4`. In Zukunft wird daran gedacht, die Erkennung des wechselseitigen Ausschlusses dahingehend zu erweitern, daß auch Ausschlußeigenschaften von Bedingungen wie z.B. `a<5` und `a+10>16` gefunden werden.
Neben dem wechselseitigen Ausschluß wird die *Implikation von Bedingungen* zur Vereinfachung von Ausdrücken verwendet. Wiederum werden nur Vergleiche von Vektoren mit Konstanten betrachtet. So gestatten z.B. `(ir[0:2]=101B3)` und `(ir[0]=1B1)` folgende Implikationen:

- `(ir[0:2]=101B3)` $\Rightarrow$ `(ir[0]=1B1)` und
- $\neg$`(ir[0]=1B1)` $\Rightarrow$ $\neg$`(ir[0:2]=101B3)`.

Die Umkehrung ist jedoch nicht möglich:

- `(ir[0]=1B1)` $\not\Rightarrow$ `(ir[0:2]=101B3)` und
- $\neg$`(ir[0:2]=101B3)` $\not\Rightarrow$ $\neg$`(ir[0]=1B1)`.

Eine Bedingung wird vereinfacht, indem sowohl alle vorausgehenden Bedingungen, als auch der wechselseitige Ausschluß und die Implikation von Bedingungen berücksichtigt werden. Unter Verwendung binärer Entscheidungsdiagramme [Bry86] ist eine effiziente Berechnung möglich. Ist das Ergebnis $True$ oder $False$, wurde ein logisch unmöglicher Pfad gefunden, so daß die if-then-else-Anweisung durch den then- oder else-Zweig ersetzt werden kann. Andernfalls wird unter Einbeziehung algebraischer Gesetze ein minimaler textueller Ausdruck abgeleitet. Da Heuristiken verwendet werden, ist der abgeleitete Ausdruck mit dem ursprünglichen zu vergleichen, um denjenigen auszuwählen, der den geringeren Aufwand verursacht. Der Aufwand eines booleschen Terms ergibt sich durch die Anzahl auftretender Variablen und Operationen.
Datenabhängigkeiten zwischen den vorausgehenden Bedingungen und sequentiellen Zuweisungen können die Vereinfachung nachfolgender Anweisungen verhindern. Aus diesem Grund werden Abhängigkeiten durch Techniken wie *Forwarding* oder *boolesche Reduktion* gelöst. Der wechselseitige Ausschluß und die Implikation von Bedingungen sind davon nicht betroffen, da diese Eigenschaften pfadunabhängig sind.

7.3 Anwendung von Transformationen

Algorithmische Beschreibungen können durch Anwendung korrektheitserhaltender Transformationen vereinfacht werden [Hin98a]. Die formalen Transformationen (siehe Abschnitt 4.6) erlauben es, gleiche Pfade in verschachtelten if-then-else-Anweisungen zu eliminieren. Während die Axiome A0, A4 und A5 Strukturen vereinfachen, dienen A1, A2 und A3 dazu, bedingte Ausführungspfade von if-then-else-Ausdrücken umzuordnen. Durch Anwendung der Transformationen entstehen komplexere Bedingungen, die sich aber häufig vereinfachen lassen.

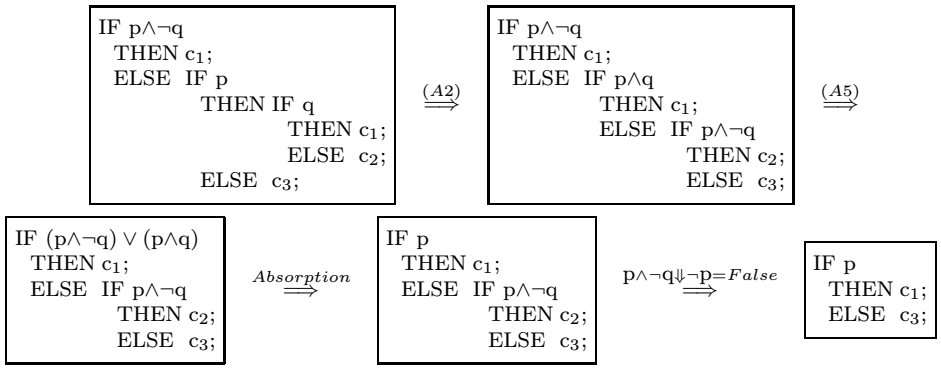


Abb. 7.2: Beispiel für eine Vereinfachung

Beispiel 7.1 *Bei dem in Abbildung 7.2 dargestellten Beispiel werden die Bedingungen mit Hilfe des Absorptions-Gesetzes (siehe Abschnitt 7.4.1) vereinfacht. Durch Anwendung des Restrict-Operators ⇓ [CM91] (siehe Abschnitt 7.4.3) kann ein logisch unmöglicher Pfad entdeckt und eliminiert werden.*

7.4 Binäre Entscheidungsdiagramme

7.4.1 Darstellung boolescher Bedingungen

Die Bedingungen repräsentieren boolesche Funktionen $f : B^n \rightarrow B$, die sich durch binäre Entscheidungsdiagramme [Bry86] darstellen lassen [HRE00]. *Ordered Binary Decision Diagrams (OBDD's)* ermöglichen eine kanonische und kompakte Darstellung boolescher Funktionen. Berechnet werden die Entscheidungsdiagramme unter Verwendung des *Multiple-Domain Decision Diagram Package* (TUDD-Package) [Hör97, Hör98a, Hör98b], das am Lehrstuhl entwickelt wurde. Der Berechnungsaufwand von Techniken ([BDL98], *BMD [BC95] oder Vektoren von OBDD's [REH99]), die es erlauben, Ausdrücke über Bitvektoren darzustellen und zu manipulieren, ist nicht akzeptabel und zum Zweck der Vereinfachung auch nicht notwendig. Die betrachteten Bedingungen bestehen aus Ausdrücken über Bit-Vektoren, die konjunktiv oder disjunktiv verknüpft sind. Sie werden

als OBDD dargestellt, indem jede boolesche Variable und jeder Teilausdruck, in dem Bit-Vektoren oder binäre Operationen, abgesehen von $\wedge$, $\vee$ und $\neg$, verwendet werden, mit einem Knoten eines OBDD assoziiert wird. Im folgenden wird ein mit einem Knoten assoziierter Ausdruck als *atomare Formel* bezeichnet.

Beispiel 7.2 *Abbildung 7.3 stellt eine Bedingung als OBDD dar. Die boolesche Variable* `flush` *und die Ausdrücke* `ir[0:2]=101B5` *und* `temp=0B16` *sind atomare Formeln, die von Knoten des Graphen repräsentiert werden.*

a) `flush`$\vee$`((ir[0:5]=101B6)`$\wedge\neg$`(temp=0B16))` b)

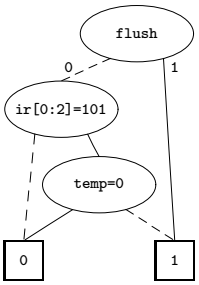


Abb. 7.3: Darstellung eines booleschen Ausdrucks als OBDD

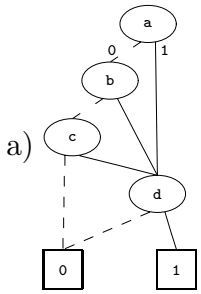
Entweder können Bedingungen entschieden werden, der berechnete Graph besteht aus einem Blatt, das die logische 0 oder 1 repräsentiert, oder ein äquivalenter Ausdruck wird von dem Entscheidungsdiagramm abgeleitet.
Heuristiken, die algebraische Gesetze benutzen, werden auf den abgeleiteten Ausdruck angewendet, um Redundanzen, die sich aus der textuellen Darstellung ergeben, zu eliminieren. Die folgenden algebraischen Gesetze erlauben es, den Aufwand boolescher Formeln zu reduzieren:

- $a \vee a = a$, $a \wedge a = a$ (Idempotenz),
- $\neg(\neg a) = a$ (Involution),
- $a \vee (\neg a \wedge b) = a \vee b$, $a \wedge (\neg a \vee b) = a \wedge b$, $a \vee (a \wedge b) = a$, $a \wedge (a \vee b) = a$ (Absorption),
- $(a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$, $(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$ (Distributivität) und
- $\neg a \wedge \neg b = \neg(a \vee b)$, $\neg a \vee \neg b = \neg(a \wedge b)$ (De Morgan).

Während die Assoziativität und die Kommutativität das Umordnen erlauben, ermöglicht das Komplement ($a \wedge \neg a = 0$ oder $a \vee \neg a = 1$) das (teilweise) Entscheiden von Termen.

Beispiel 7.3 *In Abbildung 7.4 wird aus einem binären Entscheidungsdiagramm ein textueller Ausdruck abgeleitet, indem alle erfüllenden Belegungen* `(a`$\wedge$`d)`*,*

(¬a∧b∧d) *und* (¬a∧¬b∧c∧d) *disjunktiv verknüpft werden. Der abgeleitete Ausdruck kann unter Anwendung des Distributiv- und des Absorptions-Gesetzes zu* d∧(a∨b∨c) *vereinfacht werden. Da Heuristiken zur Anwendung kommen, wird die optimale Lösung nicht immer gefunden.*



a) b) (a∧d)∨(¬a∧b∧d)∨(¬a∧¬b∧c∧d) ≡ d∧(a∨b∨c)

Abb. 7.4: Ableitung eines booleschen Ausdrucks

7.4.2 Bewertung boolescher Ausdrücke

Eine Bewertungsfunktion für boolesche Ausdrücke ist erforderlich, um unter mehreren äquivalenten Termen denjenigen auszuwählen, der die geringste Komplexität besitzt. Zur Vereinfachung boolescher Bedingungen werden binäre Entscheidungsdiagramme verwendet. Läßt sich eine Bedingung nicht entscheiden, wird ein gleichwertiger textueller, unter Berücksichtigung algebraischer Gesetze vereinfachter Ausdruck aus dem Graphen abgeleitet. In vielen Fällen kann ein weniger komplexer Ausdruck einer mehrstufigen Logik erzeugt werden, in seltenen Fällen scheitert der Versuch einer Vereinfachung, da es sich bei den angewendeten Verfahren um Heuristiken handelt. Aus diesem Grund ist es erforderlich, eine Bewertungsfunktion für boolesche Ausdrücke einzuführen, anhand deren entschieden werden kann, ob der ursprüngliche oder der abgeleitete Term weniger aufwendig ist.
Diese Bewertungsfunktion soll unabhängig von einer bestimmten Zielarchitektur sein, da algorithmische Beschreibungen zu vereinfachen sind, und mit wenig Aufwand berechnet werden können. Im folgenden wird der Aufwand eines booleschen Ausdrucks durch die Anzahl der auftretenden Variablen und der verwendeten Operationen bestimmt.

Definition 7.1 (Aufwand boolescher Ausdrücke)
Sei $\mathcal{EX}$ die Menge aller booleschen Ausdrücke und $\mathcal{AF} \subseteq \mathcal{EX}$ die Menge aller atomarer Formeln, dann berechnet die Funktion $Cost : \mathcal{EX} \rightarrow \mathcal{N}_0$ den Aufwand eines Ausdrucks $ex \in \mathcal{EX}$.

$$Cost(ex) = \left(\sum_{af \in \mathcal{AF}} Auftreten\ von\ af\ in\ ex \right) + \left(\sum_{o \in \{\wedge,\vee,\neg\}} Auftreten\ von\ o\ in\ ex \right)$$

Ein boolescher Term $ex \in \mathcal{EX}$ wird durch $ex' \in \mathcal{EX}$ ersetzt, wenn $ex' \equiv ex$ und $Cost(ex') < Cost(ex)$ erfüllt sind.

Beispiel 7.4 *Der boolesche Ausdruck* $ex = (a \wedge b \wedge c) \vee (a \wedge c) \vee (a \wedge d)$ *wird zu* $ex' = a \wedge (c \vee d)$ *vereinfacht.* ex' *ersetzt* ex*, da der Aufwand für den minimierten Ausdruck lediglich* $cost(ex') = 5$ *beträgt, der des ursprünglichen aber bei* $cost(ex) = 13$ *liegt.*

7.4.3 Vereinfachung verschachtelter Verzweigungen

Die vorausgehenden Bedingungen auf einem Pfad, der wechselseitige Ausschluß und die Implikation von Bedingungen schränken die Gültigkeit nachfolgender Bedingungen ein. Im folgenden wird vorausgesetzt, daß **keine Datenabhängigkeiten** zwischen den vorausgehenden Bedingungen und sequentiellen Zuweisungen bestehen. In Abschnitt 7.5 werden sequentielle Datenabhängigkeiten berücksichtigt und gelöst. Um den jeweiligen Ausführungszweig einer Verzweigung zu bestimmen, wird eine Bedingung ausgewertet. Das Entscheiden einer Bedingung schränkt den Wertebereich nachfolgender Operationen ein. Sind Verzweigungen verschachtelt, können Bedingungen tieferer Ebenen unter Einbeziehung von Bedingungen höherer Ebenen vereinfacht oder sogar entschieden werden.

Beispiel 7.5 *Abbildung 7.5 gibt ein Beispiel für eine baumartige Struktur und gibt die entlang eines Pfades gültigen Bedingungen an.* $\mathtt{p}_1$ *bis* $\mathtt{p}_4$ *repräsentieren einen beliebigen Ausdruck mit booleschem Ergebnis.*

```
1   IF p1                          |  1   True
2     THEN ...                     |  2   p1
3          IF p2                   |  3   p1
4            THEN ...              |  4   p1∧p2
5            ELSE ...              |  5   p1 ∧ ¬p2
6     ELSE ...                     |  6   ¬p1
7          IF p3                   |  7   ¬p1
8            THEN ...              |  8   ¬p1∧p3
9                 IF p4            |  9   ¬p1∧p3
10                  THEN ...       |  10  ¬p1∧p3∧p4
11                  ELSE ...       |  11  ¬p1∧p3 ∧ ¬p4
12           ELSE ...              |  12  ¬p1 ∧ ¬p3
```

Abb. 7.5: Baumartige Strukturen verschachtelter if-then-else-Strukturen

Da alle auftretende Bedingungen als Entscheidungsdiagramme repräsentiert werden, können die Einschränkungen, die auf einem Pfad gelten, durch den *Restrict-Operator* $\Downarrow$ [CMB91] einbezogen werden. Eine boolesche Funktion f kann durch die Funktion $f \Downarrow g$ ersetzt werden, falls es genügt, daß $f \Downarrow g$ nur in einem bestimmten, durch eine Einschränkung g gegebenen Bereich mit der ursprünglichen

Funktion f übereinstimmt. Für $f \Downarrow g$ gilt $(f \wedge g) \leq (f \Downarrow g) \leq (f \vee \neg g)$.

$$f \Downarrow g = \begin{cases} 0 & :(f = \neg g) \vee (g = 0) \\ 1 & :(f = g) \\ f & :(f = 0) \vee (f = 1) \vee (g = 1) \\ f_{x_i} \Downarrow g_{x_i} & :(g_{\overline{x_i}} = 0) \\ f_{\overline{x_i}} \Downarrow g_{\overline{x_i}} & :(g_{x_i} = 0) \\ f \Downarrow \exists x_i : g & :x_i \text{ tritt nicht in } f \text{ auf} \\ (\neg x_i \wedge (f_{\overline{x_i}} \Downarrow g_{\overline{x_i}})) \vee & \\ \quad (x_i \wedge (f_{x_i} \Downarrow g_{x_i})) & :sonst \end{cases} \tag{7.1}$$

Formel 7.1 ist entnommen [MT98] und berechnet $f \Downarrow g$ aus den Graphen f und g mit der Variablen-Ordnung $x_0 < ... < x_n$ und x_i, $0 \leq i \leq n$, als Variable minimaler Ordnung, die in f oder g auftritt.

Beispiel 7.6 *In dem Beispiel aus Abbildung 7.6 kann die Funktion* $f = (b \wedge c \wedge e) \vee (c \wedge d \wedge e) \vee (b \wedge c \wedge d)$*, die durch* $g = a \wedge b \wedge e$ *beschränkt ist, unter Anwendung der Formel 7.1 zu* $f \Downarrow g = c$ *vereinfacht werden.*

$$\begin{aligned} & f \Downarrow g \\ & \qquad \boxed{a < b < c < d < e} \\ & = (b \wedge c \wedge e) \vee (c \wedge d \wedge e) \vee (b \wedge c \wedge d) \Downarrow (a \wedge b \wedge e) \\ & \qquad \boxed{a \text{ tritt in } f \text{ nicht auf}} \\ & = (b \wedge c \wedge e) \vee (c \wedge d \wedge e) \vee (b \wedge c \wedge d) \Downarrow (b \wedge e) \\ & \qquad \boxed{g_{(b=0)} = 0} \\ & = (c \wedge e) \vee (c \wedge d \wedge e) \vee (c \wedge d) \Downarrow e \\ & \qquad \boxed{\text{sonst: } f_{(c=1)} = (e \vee d) \wedge f_{(c=0)} = 0} \\ & = c \wedge ((e \vee d) \Downarrow e) \\ & \qquad \boxed{\text{sonst: } f_{(d=1)} = 1 \wedge f_{(d=0)} = e} \\ & = c \wedge (d \vee (\neg d \wedge (e \Downarrow e))) \\ & \qquad \boxed{f = g = 1} \\ & = c \wedge (d \vee \neg d) \\ & \qquad \boxed{\text{minimale Darstellung als OBDD}} \\ & = c \end{aligned}$$

Abb. 7.6: Berechnung von $f \Downarrow g$

Alle Bedingungen verschachtelter if-then-else-Strukturen werden mit Hilfe des Restrict-Operators vereinfacht, indem das Produkt aller vorangehender Bedingungen als Beschränkung verwendet wird.

Beispiel 7.7 *Für das Beispiel aus Abbildung 7.5 werden die Graphen* $p_2 \Downarrow p_1$*,* $p_3 \Downarrow \neg p_1$ *und* $p_4 \Downarrow (\neg p_1 \wedge p_3)$ *berechnet. Ist es nicht möglich, die Bedingungen unter Einbeziehung vorausgehender Bedingungen zu entscheiden, wird ein textueller Ausdruck abgeleitet. Die Substitution einer Bedingung durch den abgeleiteten Term ist wiederum davon abhängig, ob der abgeleitete Ausdruck geringeren Aufwand als der ursprüngliche verursacht.*

Die dargestellte Technik kann auch zur Vereinfachung von Multiplexer-Funktionen verwendet werden, d.h. Strukturen, in denen n Steuerleitungen eins aus 2^n Datenwörter auswählen. Jedes Steuerungsbit repräsentiert eine boolesche Formel. Da eine bestimmte Kombination von Bit ein spezielles Datenwort auswählt, können dieses unter der Berücksichtigung des Produkts der Steuerbit minimiert werden.

Beispiel 7.8 *Der 4;1-Multiplexer* `mux(1#(a∧b)#(a∧c)#b,a#b)` *kann durch Anwendung des Restrict-Operators* `mux((1⇓ ¬a∧¬b)#(a∧b⇓ ¬a∧b)#(a∧c⇓a∧¬b)#(b⇓a∧b),a#b)` *zu* `mux(1#0#c#1,a#b)` *vereinfacht werden. Das Zeichen* `#` *bezeichnet die Konkatenation von Bit zu einem booleschen Vektor.*

Die Größe des Graphen $f \Downarrow g$, die durch die Anzahl der Graphknoten gemessen wird, ist nicht nur kleiner gleich der Größe von f, sondern auch der von $f \downarrow g$. $\downarrow$ bezeichnet den *Constrain-Operator* [CM91]. Tritt eine Variable in g, jedoch nicht in f auf, kann es in Abhängigkeit von der Entwicklungsreihenfolge passieren, daß diese Variable auch in $f \downarrow g$ vorkommt. In diesem Fall ist der durch den Restrict-Operator berechnete Graph kompakter, da Variablen, die ausschließlich in g auftreten, existentiell ausquantifiziert werden.

7.4.4 Wechselseitiger Ausschluß von Bedingungen

Die binären Entscheidungsdiagramme berücksichtigen nicht den wechselseitigen Ausschluß, da die Bedingungen nicht bitweise dargestellt werden. Aus diesem Grund wird ein zusätzliches Prädikat eingeführt, das den wechselseitigen Ausschluß aller auftretenden Bedingungen formuliert [HRE00]. Dieses Prädikat gilt pfadunabhängig, so daß sequentielle Abhängigkeiten unberücksichtigt bleiben.

Definition 7.2 (Wechselseitiger Ausschluß)
Sei $\overline{\mathcal{AF}}$ eine Menge sich wechselseitig ausschließender atomarer Formeln, dann berechnet $mutex(\overline{\mathcal{AF}})$ den wechselseitigen Ausschluß.

$$mutex(\overline{\mathcal{AF}}) = \left[\bigvee_{af \in \overline{\mathcal{AF}}} af \wedge \left(\bigwedge_{af' \in \overline{\mathcal{AF}} \setminus \{af\}} \overline{af'} \right) \right] \vee \left(\bigwedge_{af \in \overline{\mathcal{AF}}} \overline{af} \right) \tag{7.2}$$

Die Funktion $MUTEX(\mathcal{B})$ berechnet den wechselseitigen Ausschluß einer Beschreibung $\mathcal{B}$ für alle Mengen sich wechselseitig ausschließender atomarer Formeln $\overline{\mathcal{AF}}$ in $\mathcal{B}$.

$$MUTEX(\mathcal{B}) = \bigwedge_{\overline{\mathcal{AF}}\ in\ \mathcal{B}} mutex(\overline{\mathcal{AF}}) \tag{7.3}$$

Ein zusätzlicher OBDD repräsentiert den wechselseitigen Ausschluß gemäß Formel (7.3).

Beispiel 7.9 *In Abbildung 7.7 wird der wechselseitige Ausschluß von fünf Prädikaten dargestellt. Zum einen ist entweder* `pc[3]`$\wedge$`pc[5]` *oder* `pc=0` *erfüllbar, zum anderen schließen sich* `ir[1:0]=00B2`, `ir[1]` *und* `ir[0:2]=101B3` *gegenseitig aus. Abbildung 7.7(a) stellt den OBDD dar, der den wechselseitigen Ausschluß für* `pc` *repräsentiert, 7.7(b) den, der das Ausschlußkriterium für* `ir` *formuliert. Eine Beschreibung wird vereinfacht, indem das Produkt der beiden OBDD's berücksichtigt wird. Dieses Entscheidungsdiagramm ist Abbildung 7.7(c) zu entnehmen.*

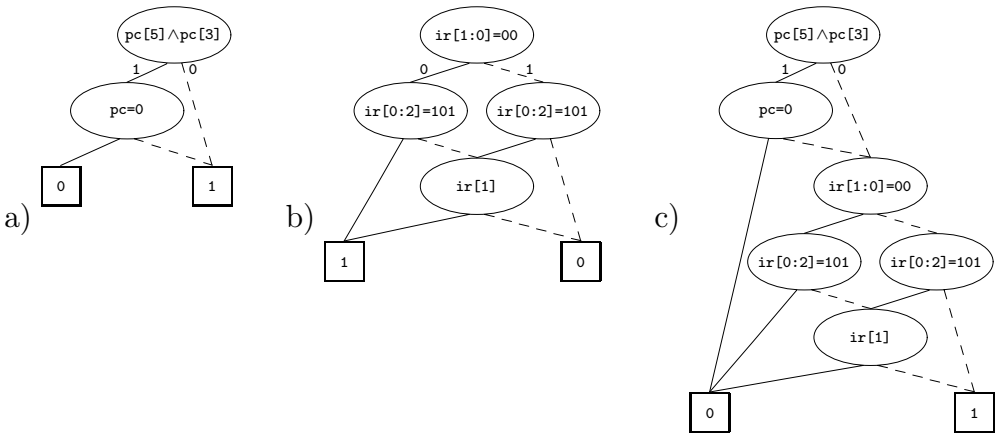


Abb. 7.7: Berechnung eines OBDD's für den wechselseitigen Ausschluß

Der wechselseitige Ausschluß gestattet es, boolesche Terme zu vereinfachen oder auszuwerten. Wenn die booleschen Ausdrücke a und b sich wechselseitig ausschließen, gilt:

- $a \wedge b \equiv False$ da a b ausschließt,
- $a \wedge \neg b \equiv a$ da a in $\neg b$ enthalten ist,
- $a \vee \neg b \equiv \neg b$ da $\neg b$ a enthält,
- $a \Downarrow b \equiv False$ da b a ausschließt und
- $\neg a \Downarrow b \equiv True$ da b $\neg a$ impliziert.

In einer Beschreibung wird der Graph jeder Bedingung unter dieser Einschränkung berechnet. Innerhalb verschachtelter if-then-else-Strukturen wird das Produkt des wechselseitigen Ausschlusses und der vorausgehenden Bedingungen als Restriktion verwendet, um nachfolgende Ausdrücke zu vereinfachen.
Unter Berücksichtigung des wechselseitigen Ausschlusses werden für das Beispiel aus Abbildung 7.5 die folgenden Graphen berechnet: $p_1 \Downarrow MUTEX(\mathcal{B})$, $p_2 \Downarrow$

$(p_1 \wedge MUTEX(\mathcal{B}))$, $p_3 \Downarrow (\neg p_1 \wedge MUTEX(\mathcal{B}))$ und $p_4 \Downarrow (\neg p_1 \wedge p_3 \wedge MUTEX(\mathcal{B}))$. Ob ein Ausdruck ersetzt wird, ist jedoch wiederum abhängig vom Aufwand. Der wechselseitige Ausschluß gestattet im Unterschied zu den vorangehenden Abschnitten auch eine Vereinfachung von p_1.

7.4.5 Implikation von Bedingungen

Auch die Implikation von Bedingungen kann ausgenutzt werden, boolesche Bedingungen zu vereinfachen. Sie wird durch Einführen eines weiteren, pfadunabhängigen Prädikats berücksichtigt [HRE00], so daß sequentielle Abhängigkeiten nicht beachtet werden müssen.

Definition 7.3 (Implikation von Bedingungen)
Sei $\mathcal{AF}$ eine Menge atomarer Formeln, $af \in \mathcal{AF}$ ein daraus ausgewählter Term und $\mathcal{AF}_{af} \subseteq \mathcal{AF} \backslash \{af\}$ die Menge aller implizierten atomaren Formeln, dann berechnet $implies(\mathcal{AF}_{af})$ das Prädikat der Implikation.

$$implies(\mathcal{AF}_{af}) = \bigwedge_{af' \in \mathcal{AF}_{af}} af \Rightarrow af' \tag{7.4}$$

Für jede atomare Formel können in einer Beschreibung $\mathcal{B}$ implizierte Terme existieren. Die Funktion $IMPLIES(\mathcal{B})$ berechnet die Implikation aller atomaren Formeln einer Beschreibung $\mathcal{B}$.

$$IMPLIES(\mathcal{B}) = \bigwedge_{\substack{af \in \mathcal{AF}: \\ \mathcal{AF}_{af} \text{ in } \mathcal{B}}} implies(\mathcal{AF}_{af}) \tag{7.5}$$

Wie der wechselseitige Ausschluß restringiert die Implikation jede Bedingung in einer Beschreibung. Das Produkt aus $MUTEX(\mathcal{B})$, $IMPLIES(\mathcal{B})$ und allen vorausgehenden Bedingungen wird verwendet, um die Bedingungen innerhalb verschachtelter if-then-else-Strukturen zu vereinfachen.
Es werden für das Beispiel der Abbildung 7.5 unter Einbeziehung der Implikation folgende Graphen berechnet: $p_1 \Downarrow (MUTEX(\mathcal{B}) \wedge IMPLIES(\mathcal{B}))$, $p_2 \Downarrow (p_1 \wedge MUTEX(\mathcal{B}) \wedge IMPLIES(\mathcal{B}))$, $p_3 \Downarrow (\neg p_1 \wedge MUTEX(\mathcal{B}) \wedge IMPLIES(\mathcal{B}))$ und $p_4 \Downarrow (\neg p_1 \wedge p_3 \wedge MUTEX(\mathcal{B}) \wedge IMPLIES(\mathcal{B}))$.

7.5 Sequentielle Anweisungen

In Abschnitt 7.4.3 wurde vorausgesetzt, daß keine Datenabhängigkeiten zwischen sequentiellen Anweisungen und vorausgehenden Bedingungen bestehen. In verschachtelten if-then-else-Strukturen können jedoch beliebig viele sequentielle Anweisungen ausgeführt werden. Datenabhängigkeiten verhindern, daß die vorausgehenden Bedingungen zur Vereinfachung nachfolgender Ausdrücke verwendet werden können.

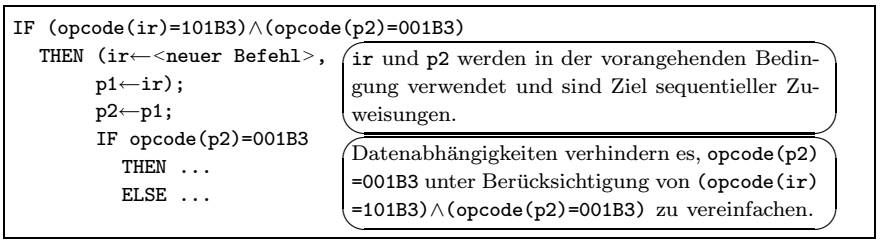


Abb. 7.8: Datenabhängigkeiten zwischen Bedingungen und Anweisungen

Beispiel 7.10 *Wird die Bedingung* (opcode(p2)=001B3) *in dem Beispiel aus Abbildung 7.8 durch*

(opcode(p2)=001B3)⇓(opcode(ir)=101B3)∧(opcode(p2)=001B3)

vereinfacht, ergibt sich ein falsches Ergebnis, da Abhängigkeiten zwischen der vorausgehenden Bedingung und den Zuweisungen ir←<neuer Befehl>, p1←ir *und* p2←p1 *bestehen.*

Die Bernsteinschen Regeln [Ber66] (siehe Abschnitt 4.7) fordern, daß die Menge der Zielvariablen der Anweisungen und die der Quellvariablen der vorausgehenden Bedingungen disjunkt sind. Wird diese Forderung nicht erfüllt, ist es entweder nicht möglich, die vorausgehenden Bedingungen bei der Vereinfachung der nachfolgenden Ausdrücke zu berücksichtigen, oder es ist notwendig, die Datenabhängigkeiten durch Methoden wie *Forwarding* (siehe Abschnitt 4.7) oder *boolesche Reduktion* [HRE00] zu lösen. Die boolesche Reduktion wird am Ende des Abschnitts definiert. Drei Fälle werden bei der Lösung von Datenabhängigkeiten unterschieden:

- **Forwarding:** Eine Variable $v \in \mathcal{V}$, die in den vorausgehenden Bedingungen verwendet wird, bekommt bei zeitgleicher Auszuführung der Zuweisung $v' \leftarrow v$ durch $v \leftarrow$ ‹neuer Wert› einen neuen Wert zugewiesen. Forwarding löst diese Datenabhängigkeit.

- **Reduktion:** Eine Variable $v \in \mathcal{V}$ wird in den vorausgehenden Bedingungen verwendet und erhält durch $v \leftarrow$ ‹neuer Wert› einen neuen Wert zugewiesen. Da der alte Wert jedoch nicht mehr in einer anderen Variablen gesichert wird, muß v in dem Ausdruck durch boolesche Reduktion eliminiert werden.

- **Konjunktion+Forwarding:** Die Variable $v \in \mathcal{V}$ tritt in den vorausgehenden Bedingungen auf, erhält aber keinen neuen Wert zugewiesen. Existiert die Zuweisung $v' \leftarrow v$, wird der ursprüngliche Ausdruck durch Konjunktion mit dem durch Forwarding erzielten verknüpft.

Beispiel 7.11 *Abbildung 7.9 zeigt die Verfeinerung der vorausgehenden Bedingung für das Beispiel aus Abbildung 7.8.*

- **Forwarding:** *Während* `ir` *einen neuen Wert zugewiesen bekommt, wird* `p1←ir` *parallel ausgeführt. Tritt* `ir` *in der vorausgehenden Bedingung auf, wird* `(opcode(p1)=101B3)∧(opcode(p2)=001B3)` *als vorausgehende Bedingung berücksichtigt.*

- **Reduktion:** *Die Variable* `p2` *bekommt einen neuen Wert durch* `p2←p1` *zugewiesen, wobei der alte Wert verloren geht. In* `(opcode(p1)=101B3)∧ (opcode(p2)=001B3)` *müssen alle Teilterme, die* `p2` *verwenden, entfallen. Das Prädikat wird zu* `(opcode(p1)=101B3)` *reduziert.*

- **Konjunktion+Forwarding:** *Da* `p2←p1` *als Zuweisung auszuführen ist, wird aus* `(opcode(p1)=101B3)` *durch Forwarding* `(opcode(p2)=101B3)`. *Als Prädikat wird* `(opcode(p1)=101B3)∧(opcode(p2)=101B3)` *berücksichtigt, weil* `p1` *keinen neuen Wert zugewiesen bekommt.*

Unter Berücksichtigung aller Datenabhängigkeiten ergibt sich als zu berücksichtigendes Prädikat `(opcode(p1)=101B3)∧(opcode(p2)=101B3)`, *so daß die nachfolgende Bedingung* `(opcode(p2)=001B3)` *ausgewertet werden kann. Der then-Zweig entfällt.*

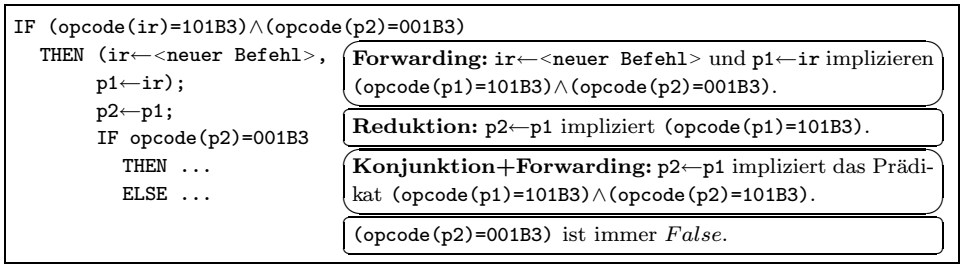


Abb. 7.9: Verfeinerung einer Bedingung

Die Funktion $Refine(ex, \mathcal{A})$ [HRE00] löst Datenabhängigkeiten zwischen einem Ausdruck ex, der die vorausgehenden Bedingungen repräsentiert, und einer Menge von Zuweisungen $\mathcal{A}$.

Definition 7.4 (Verfeinerung boolescher Ausdrücke)
Sei ex ein boolescher Ausdruck, $\mathcal{V}$ eine Menge von Variablen und $\mathcal{A}$ eine Menge von Zuweisungen. Die Funktion $S : \mathcal{EX} \rightarrow \mathcal{V}$ gibt alle in dem Ausdruck ex verwendeten Variablen an und $D : \mathcal{A} \rightarrow \mathcal{V}$ bestimmt alle Zielvariablen von Zuweisungen in $\mathcal{A}$. Die Schreibweise $ex//v' \mapsto v$ bedeutet, daß v durch v' in ex ersetzt wird.

$$Refine(ex, \mathcal{A}) = \begin{cases} Refine(ex//v'_{new} \mapsto v, \mathcal{A}) \\ \qquad :\exists v \in S(ex) \cap D(\mathcal{A}) : \exists v' \in D(\mathcal{A}) : \{v' \leftarrow v\} \in \mathcal{A} \\ Refine(Reduce(ex, \{v\}), \mathcal{A}) \\ \qquad :\exists v \in S(ex) \cap D(\mathcal{A}) : \forall v' \in D(\mathcal{A}) : \{v' \leftarrow v\} \notin \mathcal{A} \\ Refine(ex \wedge (ex//v'_{new} \mapsto v), \mathcal{A}) \\ \qquad :\exists v \in S(ex) \backslash D(\mathcal{A}) : \exists v' \in D(\mathcal{A}) : \{v' \leftarrow v\} \in \mathcal{A} \\ ex \quad :sonst \end{cases}$$

$Refine(ex, \mathcal{A})$ wird solange rekursiv aufgerufen, bis alle Datenabhängigkeiten gelöst sind und der Ausdruck sich nicht mehr verändert. Die ersetzten Variablen v' werden als v'_{new} markiert, um zu verhindern, daß sie bei einem späteren Aufruf von $Refine(ex, \mathcal{A})$ erneut substituiert werden. Wird ein Ausdruck vollständig gelöscht, gilt $Refine(ex, \mathcal{A}) = True$.

Die Prädikate, die den wechselseitigen Ausschluß und die Implikation von Bedingungen repräsentieren, werden trotz Datenabhängigkeiten nicht verfeinert, da sie pfadunabhängig sind.
Die boolesche Reduktion erlaubt es, Teilterme eines Ausdruckes, in denen bestimmte Variablen auftreten, auszuschließen.

Definition 7.5 (Boolesche Reduktion)
Sei $\mathcal{V}$ die Menge aller Variablen, sei $\mathcal{EX}$ die Menge aller Ausdrücke über $\mathcal{V}$ und $\mathcal{AF} \subseteq \mathcal{EX}$ die Menge aller atomaren Formeln, dann berechnet Reduce : $\mathcal{EX} \times \mathcal{V} \rightarrow \mathcal{EX}$ die Reduktion eines Ausdrucks. Die Funktion $S : \mathcal{EX} \rightarrow \mathcal{V}$ gibt alle in dem Ausdruck ex verwendeten Variablen an.

$$Reduce(ex, \mathcal{V}) = \begin{cases} Reduce(ex', \mathcal{V}) \wedge Reduce(ex'', \mathcal{V}) \\ \qquad\qquad :ex = ex' \wedge ex'' \ mit\ ex', ex'' \in \mathcal{EX} \\ Reduce(ex', \mathcal{V}) \vee Reduce(ex'', \mathcal{V}) \\ \qquad\qquad :ex = ex' \vee ex'' \ mit\ ex', ex'' \in \mathcal{EX} \\ \neg\overline{Reduce}(ex', \mathcal{V}) \quad :ex = \neg ex' \ mit\ ex' \in \mathcal{EX} \\ True \quad :ex \in \mathcal{AF} : \exists v \in \mathcal{V} : v \in S(ex) \\ ex \quad :sonst \end{cases}$$

Die Funktion $\overline{Reduce} : \mathcal{EX} \times \mathcal{V} \rightarrow \mathcal{EX}$ reduziert negierte Ausdrücke.

$$\overline{Reduce}(ex, \mathcal{V}) = \begin{cases} \overline{Reduce}(ex', \mathcal{V}) \wedge \overline{Reduce}(ex'', \mathcal{V}) \\ \qquad\qquad :ex = ex' \wedge ex'' \ mit\ ex', ex'' \in \mathcal{EX} \\ \overline{Reduce}(ex', \mathcal{V}) \vee \overline{Reduce}(ex'', \mathcal{V}) \\ \qquad\qquad :ex = ex' \vee ex'' \ mit\ ex', ex'' \in \mathcal{EX} \\ \neg Reduce(ex', \mathcal{V}) \quad :ex = \neg ex' \ mit\ ex' \in \mathcal{EX} \\ False \quad :ex \in \mathcal{AF} : \exists v \in \mathcal{V} : v \in S(ex) \\ ex \quad :sonst \end{cases}$$

Für $Reduce(ex, \mathcal{V})$ gilt:

1. $ex \leq Reduce(ex, \mathcal{V})$,
2. $\forall v \in \mathcal{V} : v \notin S(Reduce(ex, \mathcal{V}))$.

$Reduce(ex, \mathcal{V})$ reduziert den booleschen Ausdruck ex um die Variablen $\mathcal{V}$. Ist ex ein Produkt, entfallen die gemeinsamen Variablen. Handelt es sich bei ex dagegen um eine Summe, wird der ganze Ausdruck gelöscht. Eine Negation dreht unter Berücksichtigung der De Morgan-Gesetze das Verfahren um.

$$
\begin{aligned}
& Reduce((a \wedge d) \vee (b \wedge c \wedge \neg(a \wedge d)), \{c, d\}) \\
& \qquad\qquad \boxed{\text{Disjunktion, Rekursion}} \\
&= Reduce(a \wedge d, \{c, d\}) \vee Reduce(b \wedge c \wedge \neg(a \wedge d), \{c, d\}) \\
& \qquad\qquad \boxed{\begin{array}{l} a \text{ bleibt erhalten wegen Konjunktion} \\ c \text{ enfällt wegen Konjunktion, Rekursion} \end{array}} \\
&= a \vee (b \wedge Reduce(\neg(a \wedge d), \{c, d\})) \\
& \qquad\qquad \boxed{\text{Reduktion durch } \overline{Reduce} \text{ wegen Negation}} \\
&= a \vee (b \wedge \neg\overline{Reduce}(a \wedge d, \{c, d\})) \\
& \qquad\qquad \boxed{a \text{ entfällt wegen Konjunktion}} \\
&= a \vee (b \wedge \neg False) \\
& \qquad\qquad \boxed{\text{minimale Darstellung}} \\
&= a \vee b
\end{aligned}
$$

Abb. 7.10: Beispiel für die Reduktion eines booleschen Ausdrucks

Beispiel 7.12 *Ein Beispiel für die Reduktion eines booleschen Ausdrucks gibt die Abbildung 7.10. Der Term $(a \wedge d) \vee (b \wedge c \wedge \neg(a \wedge d))$ wird um die Variablen c und d zu $a \vee b$ reduziert. Es gilt $(a \wedge d) \vee (b \wedge c \wedge \neg(a \wedge d)) < a \vee b$ und die Variablen c sowie d sind entfallen.*

7.6 Der Algorithmus Simplify

Der Algorithmus *Simplify* vereinfacht azyklische Sequenzen. Als $g_{restrict}$ wird das Produkt aus dem wechselseitigen Ausschluß (siehe Abschnitt 7.4.4) und der Implikation von Bedingungen (siehe Abschnitt 7.4.5), das während der Abarbeitung von *Simplify* ermittelt wird, übergeben. Die sukzessive Erweiterung des Produkts bleibt jedoch zur Vereinfachung der Darstellung in Algorithmus 7.1 unberücksichtigt.
Identische Pfade in verschachtelten if-then-else-Strukturen werden während einer Vorverarbeitung durch die Anwendung korrektheitserhaltender Transformationen (siehe Abschnitt 7.3) eliminiert.

Algorithmus 7.1 Simplify

input *eine Folge von Anweisungen S_1;...;S_n; und das Entscheidungsdiagramm $g_{restrict}$;*

1. **for** $i := 1$ **to** n **do**
2. **if** S_i *ist eine if-then-else-Anweisung* **then**
3. *Berechne* $C_{S_i} \Downarrow g_{restict}$;
4. **if** $C_{S_i} \Downarrow g_{restict} = TRUE$ **then**
5. S_i :=*Simplify*(S_i^T,$g_{restrict}$);
6. **elsif** $C_{S_i} \Downarrow g_{restict} = FALSE$ **then**
7. S_i :=*Simplify*(S_i^E,$g_{restrict}$);
8. **elsif** $Cost(C_{S_i} \Downarrow g_{restict}) < Cost(C_{S_i})$ **then**
9. S_i := *IF* $C_{S_i} \Downarrow g_{restict}$
10. *THEN Simplify*(S_i^T,$g_{restrict} \wedge C_{S_i}$);
11. *ELSE Simplify*(S_i^E,$g_{restrict} \wedge \neg C_{S_i}$);
12. **else**
13. S_i := *IF* C_{S_i}
14. *THEN Simplify*(S_i^T,$g_{restrict} \wedge C_{S_i}$);
15. *ELSE Simplify*(S_i^E,$g_{restrict} \wedge \neg C_{S_i}$);
16. **fi**;
17. **else**
18. **foreach** $S_{ij} \in S_i$ *ist eine if-then-else-Anweisung* **do**
19. *Simplify*(S_{ij},$g_{restrict}$);
20. **od**
21. **let** $g_{restrict} = Refine(g_{restrict}, S_i)$;
22. **fi**;
23. **od**;

return *die vereinfachte Folge von Anweisungen S_1;...;S_n;*

Besteht die Anweisung S_i aus mehreren parallelen Ausdrücken, wird *Simplify* für jede if-then-else Anweisung (Zeile 18 bis 20) rekursiv aufgerufen. If-then-else-Anweisungen werden in den Zeilen 3 bis 16 vereinfacht. Alle vorausgehenden Bedingungen werden durch Anwendung des Restrict-Operators (siehe Abschnitt 7.4.3) in der Zeile 3 berücksichtigt. Kann $C_{S_i} \Downarrow g_{restict}$ entschieden werden bzw. entspricht der Graph dem logischen $True$ oder $False$, handelt es sich um einen logisch unmöglichen Pfad. S_i wird durch die Anweisungen des then- S_i^T oder else-Zweiges S_i^E (Zeile 5 oder 7) ersetzt.
Unter der Voraussetzung, daß $Cost(C_{S_i} \Downarrow g_{restict}) < Cost(C_{S_i})$ gilt, wird C_{S_i} in der Zeile 9 durch $C_{S_i} \Downarrow g_{restict}$ ersetzt. Durch rekursives Aufrufen von *Simplify*

werden die Anweisungen S_i^T (Zeilen 5, 10, 14) bzw. S_i^E (Zeilen 7, 11, 15) vereinfacht. Ist $C_{S_i} \Downarrow g_{restict}$ weder $True$ noch $False$, wird C_{S_i} als vorausgehende Bedingung berücksichtigt, indem das Produkt aus $g_{restrict}$ und C_{S_i} bzw. $\neg C_{S_i}$ als Parameter übergeben wird. Treten Datenabhängigkeiten zwischen den vorausgehenden Bedingungen und Anweisungen auf, wird $g_{restrict}$ in Zeile 21 verfeinert.

a)

```
ir←data+1;
p0[0:1]←ir[0:1];
IF p0[0:1]=00B2
   THEN  address←stack+offset1;
         z←p0[0];
         IF ¬z
            THEN  maddr←address;
            ELSE  maddr←address+offset2;
ELSIF p0[0:1]=01B2
   THEN  address←stack+1;
         z←p0[1];
         IF ¬z
            THEN  maddr←address;
            ELSE  maddr←address+offset2;
ELSIF p0[0:1]=10B2
   THEN  address←offset1;
         z←p0[1];
         IF ¬z
            THEN  maddr←address;
            ELSE  maddr←address+offset2;
ELSE
         address←0B16;
         z←p0[0];
         IF ¬z
            THEN  maddr←address;
            ELSE  maddr←address+offset2;
```

Der else-Zeig entfällt, da nach Verfeinern (p0[0:1]=00B2)∧(z=0B1) gilt.

(p0[0:1]=01B2)∧(z=1B1) ergibt sich nach Verfeinern, so daß der then-Zweig entfällt.

Wegen (p0[0:1]=10B2)∧(z=1B1) entfällt der else-Zweig.

¬(p0[0:1]=00B2)∧¬(p0[0:1]=01B2)∧¬(p0[0:1]=10B2)∧(z=1B1) ist erfüllt, so daß der then-Zweig entfällt.

b)

```
ir←data+1;
p0[0:1]←ir[0:1];
IF p0[0:1]=00B2
   THEN  address←stack+offset1;
         z←p0[0];
         maddr←address;
ELSIF p0[0:1]=01B2
   THEN  address←stack+1;
         z←p0[1];
         maddr←address+offset2;
ELSIF p0[0:1]=10B2
   THEN  address←offset1;
         z←p0[1];
         maddr←address;
ELSE
         address←0B16;
         z←p0[0];
         maddr←address+offset2;
```

Abb. 7.11: Beispiel für eine Vereinfachung

Abbildung 7.11 gibt ein Beispiel für das Entfernen logisch unmöglicher Pfade. Die Beschreibung einer [Ber91] entnommenen, vereinfachten Adreßberechnungseinheit ist in Abbildung 7.11(a) dargestellt. Das Ausführen von *Simplify* erzeugt die Beschreibung 7.11(b). Auf jedem Pfad kann die Bedingung ¬z entschieden werden, da in z eine Konstante gespeichert ist. Einerseits helfen die vorausgehenden Bedingungen sowie deren Verfeinerung und andererseits der wechselseitige Ausschluß und die Implikation von Bedingungen, falsche Pfade zu entdecken und zu eliminieren.

7.7 Experimentelle Ergebnisse

Die vorgestellten Techniken werden eingesetzt, um Beschreibungen von Prozessoren mit Pipelining zu vereinfachen, die durch Anwenden korrektheitserhaltender Transformationen [Hin98a, EHR98, HER99, EHR99] automatisch von einer sequentiellen Spezifikation abgeleitet werden. *Simplify* wurde während der Kon-

struktion einer Beschreibung eines DLX-Prozessors mit Pipelining [HP96], einer vereinfachten Alpha-Architektur [Dig92] und eines PIC-Mikrocontrollers [Mic93] zur Vereinfachung eingesetzt. Tabelle 7.1 gibt hierzu einen Überblick.

Architektur		DLX	Alpha	PIC	gesamt
Anzahl Ausdrücke		5.318	1.346	5.180	11.844
Anzahl logisch unmöglicher Pfade		485	385	937	1.807
Entfallen atomarer Formeln	mit log. unmögl. Pfaden	2.796	850	6.051	9.697
	ohne log. unmögl. Pfade	1412	306	2.375	4.093
	Max. in einem Ausdruck	12	16	22	22
Entfallen von Variablen	mit log. unmögl. Pfaden	972	738	1.078	2.788
	ohne log. unmögl. Pfade	119	46	91	256
	Max. in einem Ausdruck	10	3	3	10
durchschnittl. Aufwand	vor der Vereinfachung	3,46	2,27	5,33	4,14
	nach der Vereinfachung	2,66	1,11	2,51	2,24
Aufwand in % nach der Vereinfachung	Aufwand der Ausdrücke	DLX	Alpha	PIC	gesamt
	1	88%	67%	85%	84%
	2 - 4	99%	87%	78%	87%
	5 - 9	81%	39%	58%	72%
	10 - 14	49%	17%	40%	40%
	15 - 29	19%	32%	54%	40%
	≥ 30	-	3%	21%	20%
	gesamt	77%	49%	47%	54%

Tab. 7.1: Vereinfachen boolescher Terme

Die Anwendung auf mehr als 10.000 Ausdrücke zeigt, daß der Aufwand (siehe Definition 7.1 Abschnitt 7.4.2) im Durchschnitt auf 54% gesenkt werden kann, wobei zu beachten ist, daß sich große Ausdrücke am effizientesten vereinfachen lassen. So können z.B. Ausdrücke mit mehr als 30 Variablen und Operationen auf mindestens 21% des Aufwandes reduziert werden.

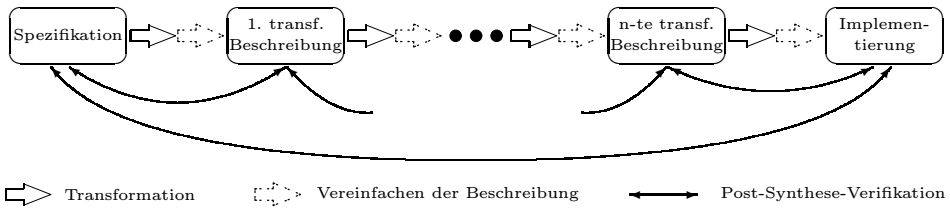


Abb. 7.12: Formal korrekte Synthese

Wie Abbildung 7.12 zeigt, wird nach jedem Zwischenschritt während des Transformationsprozesses, die Beschreibung vereinfacht. Die Konstruktion der Pipeline wäre ohne Vereinfachen nicht möglich [Hin98a, HER99]. Ohne Anwendung der

dargestellten Techniken nehmen sowohl die Größe der Beschreibung als auch die Laufzeit stark zu, da die Transformationen auf Beschreibungen angewendet werden, deren Komplexität und deren Redundanz stetig steigen.
Die Korrektheit der Vereinfachungen wird von einem unabhängigen Äquivalenzprüfer [REH99] überprüft, mit dessen Hilfe Implementierungsfehler des Transformationswerkzeugs ausfindig gemacht werden. Es besteht entweder die Möglichkeit, die Transformationen einzeln zu verifizieren oder die Korrektheit der erzeugten Pipeline bezüglich der Spezifikation nachzuweisen. Da, wie Abbildung 7.12 zu entnehmen ist, die Vereinfachungen die Synthese begleiten, werden auch sie überprüft.

7.8 Zusammenfassung

Die dargestellten Vereinfachungstechniken gestatten es, azyklische Sequenzen von Anweisungen zu vereinfachen. Einerseits werden korrektheitserhaltende Transformationen dazu verwendet, verschachtelte if-then-else-Strukturen zu vereinfachen, indem mehrfach auftretende Verzweigungspfade zusammengefaßt werden. Andererseits dienen binäre Entscheidungsdiagramme dazu, logisch unmögliche Pfade zu entdecken und Bedingungen zu minimieren.
Sequentielle Anweisungen lassen sich vereinfachen, da mit Hilfe von Verfeinerungstechniken Datenabhängigkeiten gelöst werden können. Innerhalb verschachtelter if-then-else-Anweisungen gestatten es die Bedingungen einer höheren Ebene, die Bedingungen der tieferen Ebene durch Anwendung des Restrict-Operators zu vereinfachen. Zusätzlich wird der wechselseitige Ausschluß und die Implikation von Bedingungen berücksichtigt. Kann eine Bedingung nicht entschieden werden, wird aus dem zugehörigen Graphen ein textueller Ausdruck abgeleitet, der mit Hilfe von algebraischen Gesetzen minimiert wird.
Die experimentellen Ergebnisse zeigen, daß die vorgestellten Techniken die Komplexität der Kontrollogik automatisierter Einplanungsverfahren deutlich reduzieren. Der Vereinfachungsalgorithmus ist unabhängig vom speziellen Kontext dieser Arbeit und kann aus diesem Grund auf beliebige sequentielle Programme verschiedenster Bereiche angewendet werden.

Kapitel 8

Implementierung des TUD Transformationswerkzeugs

Das *TUD Transformationswerkzeug (TUDT)* besteht aus fünf Modulen. Eine Übersicht gibt Abbildung 8.1. Das System ist in Gnu Common Lisp 2.2 (Sun Solaris) implementiert und besteht aus 19.000 Programmzeilen.

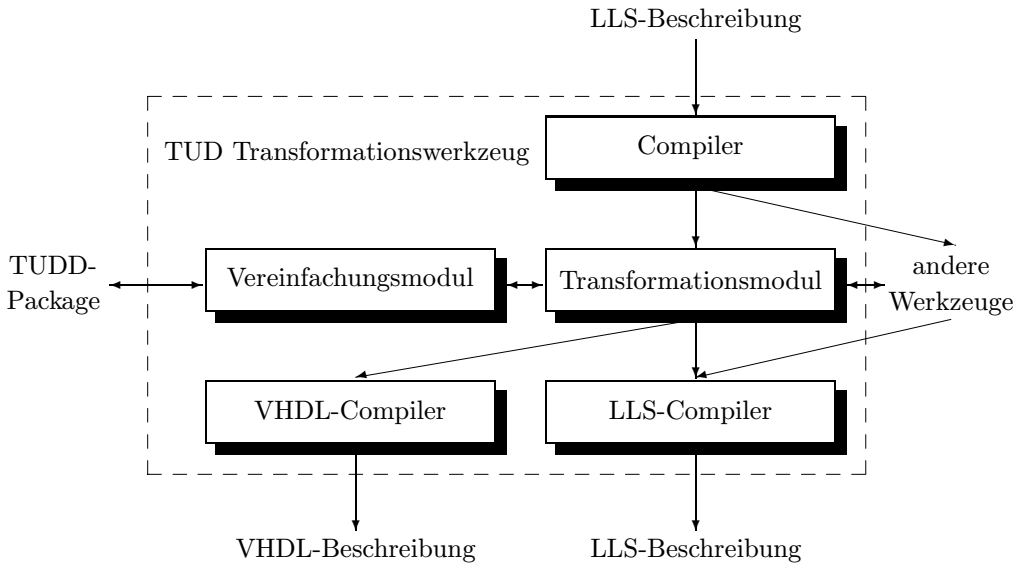


Abb. 8.1: Das TUD Transformationswerkzeug (TUDT)

Als Eingabesprache dient die im Rahmen dieser Arbeit entwickelte, experimentelle Hardwarebeschreibungssprache LLS (siehe Kapitel 3). Der *Compiler* übersetzt eine LLS-Beschreibung in eine Lisp-Datenstruktur [Hin98b], wobei sowohl die Syntax als auch die Semantik der Beschreibung überprüft werden. Darüberhinaus stellt der Compiler sicher, daß die angegebenen Transformationen realisiert werden können.

Transformationen werden von dem *Transformationsmodul* ausgeführt, welches direkt auf der Datenstruktur arbeitet. Soll ein Transformationsprozeß automatisiert werden, können die Transformationen zu *Meta-Transformationen* kombiniert werden. Das Transformationsmodul führt in diesem Fall eine Folge von Transformationen aus.
Vereinfachungen während und nach Abschluß des Transformationsprozesses (siehe Kapitel 7) führt das *Vereinfachungsmodul* durch. Boolesche Bedingungen werden unter Verwendung des am Lehrstuhl entwickelten *Multiple-Domain Decision Diagram Packages* (TUDD-Package) [Hör97, Hör98a, Hör98b] als binäre Entscheidungsdiagramme dargestellt und vereinfacht.
Als Ergebnis des Transformationsprozesses kann alternativ eine LLS-Beschreibung durch den *LLS-Compiler* oder eine VHDL-Beschreibung durch den *VHDL-Compiler* (siehe Abschnitt 3.5) generiert werden. Da vorausgesetzt wird, daß die Datenstruktur eine korrekte Beschreibung repräsentiert, werden weder Syntax noch Semantik überprüft. Bevor eine Beschreibung nach VHDL übersetzt werden kann, wird sie durch das Transformationsmodul unter Anwendung von Transformationen normalisiert. Da einige LLS-Ausdrücke wie z.B. Arrays oder Funktionen auf Arrays nur bedingt übersetzt werden können, überprüft der VHDL-Compiler, ob Einschränkungen verletzt werden.
Dient eine Beschreibung als Eingabe anderer Werkzeuge wie z.B. eines Äquivalenzprüfers [REH99, RHE99a, RHE99b] oder eines Modellprüfers [BRHE00], kann auch die Datenstruktur einer Beschreibung ausgegeben werden. Generieren die Werkzeuge eine Lisp-Datenstruktur als Ausgabe, können wiederum die einzelnen Module des *TUD Transformationswerkzeugs (TUDT)* angesprochen werden.
Da bisher noch kein Simulator existiert, können Beschreibungen nur nach einer Übersetzung nach VHDL durch kommerzielle Werkzeuge wie z.B. dem *Synopsys VHDL System Simulator* [Syn98c] simuliert werden.
Über eine graphische Oberfläche kann das Werkzeug bedient werden. Sie ist in Tk 8.0 (Sun Solaris) implementiert und besteht aus 1.100 Programmzeilen. Der am Lehrstuhl entwickelte Äquivalenzprüfer kann ebenfalls über diese Oberfläche aufgerufen werden.

Kapitel 9

Schlußbemerkungen

Die Komplexität der Schaltungsentwürfe ist in den letzten Jahren ständig gestiegen. Die immer kürzer werdende Zeitspanne des "Time-to-Market" stellt die Entwickler bei gleichzeitig ansteigenden Anforderungen an Qualität und Funktionalität der Schaltungen vor große Probleme. Bei steigender Komplexität und kürzeren Entwicklungszeiten wird es immer schwieriger, die Korrektheit der Entwürfe sicherzustellen. Aus diesem Grund kann die Korrektheit einer Schaltung im allgemeinen nicht durch Simulation alleine nachgewiesen werden.
Das in dieser Arbeit vorgestellte *TUD Transformationswerkzeug (TUDT)* ist ein interaktives Werkzeug der formalen Synthese. Der formale Kern des Werkzeugs besteht aus neun Axiomen und einer Regel, die das Umformen von Kontrollstrukturen ermöglichen. Sie sind korrektheitserhaltend im Sinne der *Pfadäquivalenz*. Sechs weitere Axiome stehen zur Verfügung, um das Zeitverhalten einer Beschreibung zu verändern. Sie sind korrekt im Sinne der *Berechnungsäquivalenz*.
Viele Werkzeuge der formalen Synthese betten den Transformationsprozeß in einen Theorembeweiser ein. Dabei ergibt sich das Problem, daß das Synthesewissen innerhalb des Theorembeweisers rekonstruiert und formalisiert werden muß. Da die Syntheseschritte auf der Ebene der elementar logischen Umformungen formulieren werden müssen, benötigt der Anwender einige Erfahrungen im Umgang mit dem jeweiligen Theorembeweiser. Aus Gründen der Bedienerfreundlichkeit und der Akzeptanz wird die Korrektheit der Ergebnisse des *TUD Transformationswerkzeugs (TUDT)* nicht durch ein Beweissystem, sondern durch Methoden der formalen Verifikation nachgewiesen. Somit ist es möglich, entweder das Ergebnis nach Durchführen einer Folge von Transformationen zu überprüfen oder jeden einzelnen Transformationsschritt zu verifizieren. Darüberhinaus ist das *TUD Transformationswerkzeugs (TUDT)* derart konzipiert, das es mit anderen Werkzeugen sowohl der formalen Verifikation als auch der Synthese kombiniert werden kann.
Ein Verfahren zur automatisierten Einplanung von dynamischen Pipelines wurde als Anwendungsbeispiel des Transformationswerkzeugs vorgestellt. Ausgehend

von einer Beschreibung eines sequentiellen Prozessors wird durch sukzessives Anwenden von Transformationen eine Implementierung als Pipeline abgeleitet. Im Gegensatz zu anderen Werkzeugen werden Abhängigkeiten von Befehlen dynamisch durch Forwarding, Einführen von Pipelineregistern und Anhalten der Pipeline gelöst. Da diese Techniken als Transformation implementiert sind, können Konflikte automatisch gelöst werden. Als Beispiele dienten eine DLX-Pipeline, ein PIC-Mikrocontroller und eine ALPHA-Risc-Architektur. Die experimentellen Ergebnisse haben gezeigt, daß die automatisch eingeplanten Pipelines mit den bekannten, manuell erzeugten Implementierungen vergleichbar sind.
Nicht nur der gesamte Transformationsprozeß der Pipeline-Synthese, sondern auch die Verifikation der Ergebnisse ließ sich automatisieren. Nach Abschluß der Transformationen konnte für alle genannten Beispiele, es handelt sich dabei um zwei- bis fünfstufige Pipelines, gezeigt werden, daß die sequentiellen Prozessorbeschreibungen und die zugehörigen Pipelinesysteme äquivalent bezüglich der Befehlsausführung sind.
Durch die Verifikation der Ergebnisse konnten mehrere Fehler in der Implementierung des Werkzeugs gefunden werden, die jedoch nicht die konzeptionelle Korrektheit der Transformationen in Frage stellten. Aufgabe der Verifikation war es, *Implementierungsfehler* des Synthesewerkzeugs zu finden. Durch die Kombination mit Methoden der formalen Verifikation, wie z.B. der symbolischen Simulation, unterstützt das *TUD Transformationswerkzeug (TUDT)* den formal korrekten Hardwareentwurf.
Die automatisierte Einplanung von Prozessoren mit Pipelining erfordert die Anwendung von Vereinfachungstechniken, die es erlauben, durch Verringern der Komplexität der Kontrollogik und Eliminieren logisch unmöglicher Pfade die Kosten des Entwurfs zu reduzieren und/oder eine Geschwindigkeitsverbesserung zu erzielen. Da sequentielle Abhängigkeiten bestehen und für eine effektive Minimierung jeder Pfad einer Beschreibung und jede auf einem Pfad gültige Bedingung berücksichtigt werden müssen, ist aufgrund des Berechnungsaufwandes eine exakte Lösung nicht möglich. Die verwendeten Heuristiken finden logisch unmögliche Pfade durch Einbeziehung des wechselseitigen Ausschlusses und der Implikation von Bedingungen, durch Berücksichtigen aller auf einem Pfad vorausgegangenen Bedingungen sowie durch Lösen sequentieller Datenabhängigkeiten und ermöglichen unter Verwendung von binären Entscheidungsdiagrammen eine effiziente Vereinfachung von Bedingungen. Das Verfahren kombiniert mehrere, bekannte Ansätzen und nutzt deren Vorteile. So wird z.B. der *Restrict-Operator* nicht zur Zustandstraversierung, sondern zur Minimierung algorithmischer Beschreibungen verwendet. Kombiniert mit den im Rahmen dieser Arbeit entwickelten Heuristiken, die aus einem binären Entscheidungsdiagramm einen textuell minimalen Ausdruck ableiten, und Verfahren, die sequentielle Abhängigkeiten lösen, ist eine effiziente Vereinfachung von algorithmischen Hardwarebeschreibungen möglich.
Neben der Synthese kann das Transformationswerkzeug auch zur Verifikation,

insbesondere von Einplanungsverfahren, verwendet werden. Die Äquivalenz von zwei Beschreibungen kann dadurch nachgewiesen werden, daß eine Folge von korrektheitserhaltenden Transformationen angegeben wird, die die eine in die andere Beschreibung umwandelt. Die Beschreibungen sind äquivalent, wenn nach Abschluß des Transformationsprozesses für jedes Segment der einen Beschreibung ein bisimulares Segment in der anderen gefunden werden kann und umgekehrt.
Das Verfahren setzt voraus, daß zwischen den Beschreibungen gewisse strukturelle Ähnlichkeiten bestehen. Aus diesem Grund wurden Ergebnisse von Einplanungsverfahren verifiziert, da solche Verfahren im allgemeinen nicht die speziellen Eigenschaften der verwendeten Funktionen beachten, so daß Datenoperationen bis auf das Einführen zusätzlicher Register oder das Durchführen von Forwarding unverändert erhalten bleiben. Die experimentellen Ergebnisse haben gezeigt, daß die Ergebnisse moderner Einplanungsverfahren wie z.B. *As Fast As Possible* oder *Pipelined Path Scheduling* mit Hilfe des transformativen Ansatzes überprüft werden können. Andere Post-Synthese-Verifikationsmethoden scheitern im allgemeinen daran, daß die Korrektheit zyklischer Beschreibungen nicht gezeigt werden kann. Der transformative Ansatz hingegen kann auf zyklische Beschreibungen angewendet werden und kann somit insbesondere den Einplanungsschritt, auch bei Anwendung moderner Einplanungsverfahren, formal verifizieren.
Das *TUD Transformationswerkzeug (TUDT)* wird mehrfach im Entwurfsfluß verwendet. Es übernimmt die Einplanungsphase, sofern kein klassisches Verfahren angewendet wird. Auch die Normalisierung von Beschreibungen bzw. die Umwandlung in einen erweiterten endlichen Automaten erfolgt unter Anwendung von Transformationen. Werden zyklische Beschreibungen verarbeitet, dient das Werkzeug zur Verifikation. Als textuelle Repräsentation wird die im Rahmen dieser Arbeit entwickelte, experimentelle Hardwarebeschreibungssprache *Language of Labelled Segments (LLS)* benutzt. Durch die Übersetzung nach VHDL können auch kommerzielle Synthesewerkzeuge wie z.B. der *Synopsys Design Compiler* [Syn98a] in den Entwurfsfluß einbezogen werden.

Im Vordergrund steht in Zukunft die Automatisierung von Syntheseabläufen. Die Anwendungsbeispiele haben gezeigt, daß sich die Einplanung von dynamischen Pipelines durch Anwendung von formal korrekten Transformationen automatisieren läßt. Das Transformationswerkzeug könnte z.B. derart erweitert werden, daß eine strukturelle Repräsentation transformativ aus einer Verhaltensbeschreibung gewonnen wird. Außerdem könnte es erforderlich werden, Techniken zu entwickeln, die eine eindeutige Invertierbarkeit von kontext-abhängigen Transformationen sicherstellen. Darüberhinaus wird daran gedacht, den transformativen Ansatz für die Verifikation mit Methoden der Post-Synthese-Verifikation zu kombinieren, so daß z.B. auch komplexere Pipelinesysteme auf Korrektheit überprüft werden können.

Literaturverzeichnis

[ABRM98] P. Ashar, S. Bhattacharya, A. Raghunathan und A. Mukaiyama. Verification of RTL generated from scheduled behavior in a high-level synthesis flow. In *Proc. International Conference on Computer Aided Design (ICCAD)*, 1998.

[AL91] M. Aagaard und M. Leeser. A formally verified system for logic synthesis. In *Proc. International Conference on Computer Design (ICCD)*, Seiten 346-350, 1991.

[BBC+94] G. Bezzi, M. Bombana, P. Cavalloro, S. Conigliaro und G. Zaza. Quantitative evaluation of formal based synthesis in ASIC design. In T. Kropf und R. Kumar (Herausgeber): *Proc. International Conference on Theorem Provers in Circuit Design (TPCD)*, Band 901 der Reihe *Lecture Notes in Computer Science*, Seiten 286-291. Springer Verlag, 1994.

[BC95] R. E. Bryant und Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1995.

[BD94] J. R. Burch und D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. Computer Aided Verification (CAV)*, Band 818 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[BDL96] C. W. Barrett, D. L. Dill und J. R. Levitt. Validity checking for combinations of theories with equality. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, Band 1166 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[BDL98] C. W. Barrett, D. L. Dill und J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1998.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. In *IEEE Transactions on Electronic Computers*, Band EC-15(5), Seiten 757-763, 1966.

[Ber91] R. A. Bergamaschi. The effects of false paths in high-level synthesis. In *Proc. International Conference on Computer Aided Design (ICCAD)*, 1991.

[BHK94] B. Brock, W. A. Hunt und M. Kaufmann. The FM9001 microprocessor proof. Technischer Bericht 86, Computational Logic Inc., 1994.

[BJ93] B. Bose und S. D. Johnson. DDD-FM9001: derivation of a verified microprocessor. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Band 683 der Reihe *Lecture Notes in Computer Science*, Seiten 191-202. Springer Verlag, 1993.

[BM88] R. S. Boyer und J. S. Moore. *A computational logic handbook*. Academic Press, 1988.

[BM97] E. Börger und S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. G. Hinchey und D. Till (Herausgeber): *ZUM'97: The Z Formal Specification Notation*, Band 1212 der Reihe *Lecture Notes in Computer Science*, Seiten 151-187. Springer Verlag, 1997.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, Band C-35(8), Seiten 677-691, 1986.

[BS99] C. Blumenröhr und V. Sabelfeld. Formal synthesis at the algorithmic level. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Band 1703 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Bur96] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 552-557, 1996.

[Cam89] R. Camposano. Behavior-preserving transformations for high-level synthesis. In *Proc. Workshop on Hardware Specification , Verification and Synthesis: Mathematical Aspects*, Band 408 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[Cam91] R. Camposano. Path-based scheduling for synthesis. In *IEEE Transactions on Computer-Aided Design*, Band 10(1), Seiten 85-93, 1991.

[CM91] O. Coudert und J. C. Madre. Symbolic computation of the valid states of a sequential machine: algorithms and discussion. In *International Workshop on Formal Methods in VLSI Design*, 1991.

[CMB91] O. Coudert, J. C. Madre und C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E. M. Clarke und R. P. Kurshan (Herausgeber): *Computer-Aided Verification (CAV) '90*, Seiten 75-84. American Mathematical Society - Association for Computing Machinery, 1991.

[CPR89] C.-M. Chu, M. Potkonjak und J. Rabaey. HYPPER: an interactive synthesis environment for high performance real time applications. In *Proc. International Conference on Computer Design (ICCD)*, Seiten 432-435, 1989.

[CT93] R. J. Cloutier und D. E. Thomas. Synthesis of pipelined instruction set processors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 583-588, 1993.

[Cyr96] D. Cyrluk. Inverting the abstraction mapping: a methodology for hardware verification. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, Band 1166 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[Der95] Derivation System Inc. *DRS: Derivation Reasoning System, 1.2.1 edition*, 1995.

[Dig92] Digital Equipment Corporation. *Alpha architecture handbook*, 1992.

[EK95] D. Eisenbiegler und R. Kumar. Formally embedding existing high level synthesis algorithms. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Band 987 der Reihe *Lecture Notes in Computer Science*, Seiten 71-83. Springer Verlag, 1995.

[EKM96] D. Eisenbiegler, R. Kumar und J. Müller. A formal model for a VHDL subset of synchronous circuits. In *Proc. Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 1996.

[Eve90] H. Eveking. Verification, synthesis and correctness-preserving transformations - cooperative approaches to correct hardware design. In M. Yoeli, editor, *Formal Verification of Hardware Design*, Seiten 17-24. IEEE Computer Society Press Tutorial, 1990.

[Eve91] H. Eveking. *Verifikation digitaler Systeme*. B.G. Teubner Stuttgart, 1991.

[FFFH90] S. Finn, M. P. Fourman, M. Francis und R. Harris. Formal system design - interactive synthesis based on computer-assisted formal reasoning. In L. J. M. Claesen, editor, *International Workshop on Applied Formal Methods For Correct VLSI Design*, Seiten 139-152. Elsevier Science Publisher B.V. (North-Holland), 1990.

[FK91] F. Feldbusch und R. Kumar. Verification of synthesized circuits at register transfer level with flow graphs. In *Proc. European Conference on Design Automation (EDAC)*, Seiten 22-26, 1991.

[Glu65] V. M. Glushko. Automata theory and formal microprogram transformations. In *Kibernetika*, Band 1(5), Seiten 1-9, 1965.

[GM93] M. Gordon und T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Har97] B. Harking. *Mikroarchitektursynthese durch Quelltexttransformationen in einer Hardwarebeschreibungssprache*. Dissertation, Universität Dortmund, 1997.

[HD92] F. K. Hanna und N. Daeche. Dependent types and formal synthesis. *Philosophical Transaction Royal Society London*, 339:121-135, 1992.

[HDL89] F. K. Hanna, N. Daeche und M. Longley. Formal synthesis of digital systems. In L. J. M. Claesen, editor, *International Workshop on Applied Formal Methods For Correct VLSI Design*, Seiten 532-548. Elsevier Science Publisher B.V. (North-Holland), 1989.

[HHL91] C.-T. Hwang, Y.-C. Hsu und Y.-L. Lin. Scheduling for functional pipelining and loop winding. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 764-769, 1991.

[HM96] R. B. Hughes und G. Musgrave. The Lambda approach to system verification. In G. De Micheli und M. Sami (Herausgeber): *Hardware/Software Co-Design*, Seiten 427-451. Kluwer Academic Publishers, 1996.

[Hoa85] C. A. R. Hoar. *Communicating Sequential processes*. Prentice Hall, 1985.

[Hör97] S. Höreth. Implementation of a multiple-domain decision diagram package. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Seiten 185-202, 1997.

[Hör98a] S. Höreth. *Effiziente Konstruktion und Manipulation von binären Entscheidungsgraphen*. Dissertation, Technische Universität Darmstadt, 1998.

[Hör98b] S. Höreth. Hybrid graph manipulation package demo, URL : http://www.rs.e-technik.tu-darmstadt.de/∼sth/demo.html. Technischer Bericht, Technische Universität Darmstadt, Institut für Datentechnik, Fachgebiet Rechnersysteme, 1998.

[HP95] J. Hallberg und Z. Peng. Synthesis under local timing constraints in the CAMAD high-level synthesis system. In *Proc. Euromicro*, 1995.

[HP96] J. L. Hennessy und D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufman, second edition, 1996.

[HU94] J. E. Hopcroft und J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, third edition, 1994.

[JB97] S. D. Johnson und B. Bose. DDD: a system for mechanized digital design derivation. Technischer Bericht 323, Indiana University, Computer Science Department, 1990, überarbeitet 1997.

[JDB95] R. B. Jones, D. L. Dill und J. R. Burch. Efficient validity checking for processor verification. In *Proc. International Conference on Computer Aided Design (ICCAD)*, 1995.

[JM97] S. D. Johnson und P. S. Miner. Integrated reasoning support in system design: design derivation and theorem proving. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Seiten 255-272, 1997.

[JS93] G. Jones und M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Mathematics of Program Construction*, Band 669 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[KBES96] R. Kumar, C. Blumenröhr, D. Eisenbiegler und D. Schmid. Formal synthesis in circuit design - a classification and survey. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, Band 1166 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[KK97] A. Kuehlmamm und F. Krohm. Equivalence checking using cuts and heaps. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 263-268, 1997.

[Kna89] D. W. Knapp. An interactive tool for register-level structure optimization. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 598-601, 1989.

[KW92] D. W. Knapp und M. Winslett. A prescriptive formal model for datapath hardware. In *IEEE Transactions on Computer-Aided Design*, Band 11(2), Seiten 158-183, 1992.

[LMM98] Th. Lock, M. Mendler und M. Mutz. Combined formal post- and presynthesis verification. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, Band 1522 der Reihe *Lecture Notes in Computer Science*, Seiten 222-236. Springer Verlag, 1998.

[Mar93] P. Marwedel. *Synthese und Simulation von VLSI-Sytemen.* Hanser Verlag, 1993.

[Mic93] Microchip Technology Inc. *Microchip data book*, 1993.

[Mic94] G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw-Hill Series in Electrical and Computer Engineering, 1994.

[Mid94] P. F. A. Middelhoek. Transformational design of digital signal processing applications. In *Proc. of the IEEE/ProRISC Workshop on Signal Processing*, Seiten 176-180, 1994.

[Mid97] P. F. A. Middelhoek. *Transformational design: an architecture independent interactive design methodology for the synthesis of correct and efficient digital systems.* Dissertation, Twente University, 1997.

[Mil89] R. Milner. *Communication and concyrrency.* Prentice Hall, 1989.

[MLD92] P. Michel, U. Lauther und P. Duzy. *The synthesis approach to digital system design.* Kluwer Academic Publishers, 1992.

[MP83] M. C. McFarland und A. C. Parker. An abstract model of behavior for hardware descriptions. In *IEEE Transactions on Computers*, Band C-32(7), Seiten 621-637, 1983.

[MP91] Z. Manna und A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer Verlag, 1991.

[MR96] P. F. A. Middelhoek und S. P. Rajan. From VHDL to efficient and first-time-right designs: a formal approach. In *ACM Transactions on Design Automation of Electronic Systems*, 1996.

[MT98] Ch. Meinel und Th. Theobald *Algorithmen und Datenstrukturen im VLSI-Design.* Springer Verlag, 1998.

[Mut97] M. Mutz. Automatic post-synthesis verification support for a high level synthesis step by using the HOL theorem proving system. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Seiten 291-308, 1997.

[NTR+98] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan und R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. In *Proc. International Conference on Computer Design (ICCD)*, 1998.

[NV98] N. Narasimhan und R. Vemuri. On the effectiveness of theorem proving guided discovery of formal assertions for a register allocator in a high-level synthesis system. In *Proc. Conference on Theorem Proving in Higer Order Logic (TPHOL)*, 1998.

[PK89a] P. G. Paulin und J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. In *IEEE Transactions on Computer-Aided Design*, Band 8(6), Seiten 661-679, 1989.

[PK89b] P. G. Paulin und J. P. Knight. Scheduling and binding algorithms for high-level synthesis. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 1-6, 1989.

[PP88] N. Park und A. C. Parker. Sehwa: a software package for synthesis of pipelines from behavioral specifications. In *IEEE Transactions on Computer-Aided Design*, Band 7(3), Seiten 356-370, 1988.

[PR93] M. Potkonjak und J. Rabaey. Exploring the algorithmic design space using high level synthesis. In *VLSI Signal Processing*, 1993.

[PRK+96] L. Pirmez, M. Rahmouni, P. Kission, A. Pedroza, A. Mesquita und A. A. Jerraya. Analysis of different protocol description styles in VHDL for high-level synthesis. In *Proc. European Design Automation Conference (EURO-DAC) with EURO-VHDL*, 1996.

[Raj95] S. P. Rajan. Correctness of transformations in high level synthesis: formal verification. In *Proc. International Conference on Computer Hardware Description Languages (CHDL)*, 1995.

[RJ95a] M. Rahmouni und A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proc. European Design Automation Conference (EURO-DAC)*, 1995.

[RJ95b] M. Rahmouni und A. A. Jerraya. PPS: a pipeline path-based scheduler. In *Proc. European Design and Test Conference*, Seiten 557-561, 1995.

[ROJ94] M. Rahmouni, K. O'Brien und A. A. Jerraya. A loop-based scheduling algorithm for hardware description languages. In *Journal of Parallel Processing Letters*, Band 4(3), Seiten 351-364, 1994.

[SP94] E. Stoy und Z. Peng. A design representation for hardware/software co-synthesis. In *Proc. Euromicro*, Seiten 192-200, 1994.

[SR95] R. Sharp und O. Rasmussen. The T-Ruby design system. In *Proc. International Conference on Computer Hardware Description Languages (CHDL)*, Seiten 587-596, 1995.

[Syn98a] Synopsys Inc. *Synopsys Design Compiler*™, 1998.

[Syn98b] Synopsys Inc. *Synopsys Behavioral Compiler*™, 1998.

[Syn98c] Synopsys Inc. *Synopsys VHDL System Simulator*™, 1998.

[TDW+88] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor und R. L. Blackburn. The system architect's workbench. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 337-343, 1988.

[vE98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, Seiten 618-623, 1998.

[Vem90a] R. Vemuri. How to prove the completeness of a set of register level design transformations. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, Seiten 207-210, 1990.

[Vem90b] R. Vemuri. On the notion of the normal form register-level structures and its applications in design-space exploration. In *Proc. European Design Automation Conference (Euro-DAC)*, Seiten 46-51, 1990.

[WT89] R. A. Walker und D. E. Thomas. Behavioral transformation for algorithmic level IC design. In *IEEE Transactions on Computer-Aided Design*, Band 8(10), Seiten 1115-1128, 1989.

Eigene Veröffentlichungen

[BRHE00] C. Blank, G. Ritter, H. Hinrichsen und H. Eveking. Formale Verifikation der Register-Allokation. In *GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2000.

[EHR99] H. Eveking, H. Hinrichsen und G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[HER99] H. Hinrichsen, H. Eveking und G. Ritter. Formal synthesis for pipeline design. In *Proc. Discrete Mathematics and Theoretical Computer Science Conference (DMTCS) and Computing: The Australasian Theory Symposium (CATS)*, Band 21, Nummer 3 der Reihe *Discrete Mathematics and Theoretical Computer Science*, Seiten 247-261. Springer Verlag, 1999.

[HRE99] H. Hinrichsen, G. Ritter und H. Eveking. Automatische Synthese und Verifikation von RISC-Prozessoren. In *GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 1999.

[HRE00] H. Hinrichsen, G. Ritter und H. Eveking. False-path elimination and simplification of sequential acyclic descriptions with complex branching logic. In *Proc. Workshop on Algorithm Architecture Adequation (AAA)*, 2000.

[REH99] G. Ritter, H. Eveking und H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Band 1703 der Reihe *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[RHE99a] G. Ritter, H. Hinrichsen und H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *Asian Computing Science Conference (ASIAN)*, Band 1742 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[RHE99b] G. Ritter, H. Hinrichsen und H. Eveking. Formale Verifikation automatisch generierter Pipelinesysteme durch symbolische Simulation. In *GI/GMM/ITG Fachtagung: Entwurf integrierter Schaltungen*, 1999.

Technische Berichte

[EHR98] H. Eveking, H. Hinrichsen und G. Ritter. Formally correct construction of pipelined processors. Technischer Bericht 98-6-1, Technische Universität Darmstadt, Institut für Datentechnik, Fachgebiet Rechnersysteme, 1998.

[Hin98a] H. Hinrichsen. Formally correct construction of a pipelined DLX architecture. Technischer Bericht 98-5-1, Technische Universität Darmstadt, Institut für Datentechnik, Fachgebiet Rechnersysteme, 1998.

[Hin98b] H. Hinrichsen. Language of Labelled Segments documentation, URL : http://www.rs.e-technik.tu-darmstadt.de/~hinni/document/index.html. Technischer Bericht, Technische Universität Darmstadt, Institut für Datentechnik, Fachgebiet Rechnersysteme, 1998.

Holger Hinrichsen

April 2000

Angaben zur Person

Geburtsdatum	29. September 1970
Geburtsort	Offenbach am Main
Staatsangehörigkeit	deutsch
eMail	hinrichsen@rs.tu-darmstadt.de
Internet	http://www.e-technik.tu-darmstadt.de/∼hinni

Werdegang

1977-1981	Grundschule, Wilhelm-Busch-Schule, Rodgau
1981-1987	Gymnasium, Georg-Büchner-Schule, Rodgau
1987-1990	Gymnasiale Oberstufe, Claus-von-Staufenberg-Schule, Rodgau
1990	Abitur
1990-1997	Studium der Informatik an der Wolfgang-von-Goethe-Universität Frankfurt
1997	Abschluß als Dipl.-Informatiker
1997-2000	Stipendiat des Graduiertenkollegs *"Intelligente Systeme für die Informations- und Automatisierungstechnik"*
ab März 2000	Mitarbeiter der Allianz Versicherungs-AG, München